Universitat Politècnica de València

Cyber-security protection techniques to mitigate memory errors exploitation

Author:
Héctor Marco Gisbert

Advisor:
Ismael Ripoll Ripoll

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
(Computer Engineering)

in the
Department of Computer Engineering

November 2015

TECHNICAL UNIVERSITY OF VALENCIA

Abstract

School of Computer Engineering
Department of Computer Engineering

Doctor of Philosophy
(Computer Engineering)

Cyber-security protection techniques to mitigate memory errors exploitation

by Héctor Marco Gisbert

Practical experience in software engineering has demonstrated that the goal of building totally fault-free software systems, although desirable, is impossible to achieve. Therefore, it is necessary to incorporate mitigation techniques in the deployed software, in order to reduce the impact of latent faults.

This thesis makes contributions to three memory corruption mitigation techniques: the stack smashing protector (SSP), address space layout randomisation (ASLR) and automatic software diversification.

The SSP is a very effective protection technique used against stack buffer overflows, but it is prone to brute force attacks, particularly the dangerous 'byte-for-byte' attack. A novel modification, named 'RenewSSP', has been proposed which eliminates brute force attacks, can be used in a completely transparent way with existing software and has negligible overheads. There are two different kinds of application for which RenewSSP is especially beneficial: networking servers (tested in Apache) and application launchers (tested on Android).

ASLR is a generic concept with multiple designs and implementations. In this thesis, the two most relevant ASLR implementations of Linux have been analysed (Vanilla Linux and PaX patch), and several weaknesses have been found. Taking into account technological improvements in execution support (compilers and libraries), a new ASLR design has been proposed, named 'ASLR-NG', which maximises entropy, effectively addresses the fragmentation issue and removes a number of identified weaknesses. Furthermore, ASLR-NG is transparent to applications, in that it preserves binary code compatibility and does not add overheads. ASLR-NG has been implemented as a patch to the Linux kernel 4.1.

Software diversification is a technique that covers a wide range of faults, including memory errors. The main problem is how to create variants, i.e. programs which have identical behaviours on normal inputs but where faults manifest differently. A novel form of automatic variant generation has been proposed, using multiple cross-compiler suites and processor emulators.

One of the main goals of this thesis is to create applicable results. Therefore, I have placed particular emphasis on the development of real prototypes in parallel with the theoretical study. The results of this thesis are directly applicable to real systems; in fact, some of the results have already been included in real-world products.

UNIVERSIDAD POLITÉCNICA DE VALENCIA

Resumen

Escuela Técnica Superior de Ingeniería Informática
Departamento de Informática de Sistemas y Computadores

Doctor en Filosofía
(Ingeniería Informática)

Técnicas de ciberseguridad para mitigar la explotación de errores de memoria

por Héctor Marco Gisbert

La creación de software supone uno de los retos más complejos para el ser humano ya que requiere un alto grado de abstracción. Aunque se ha avanzado mucho en las metodologías para la prevención de los fallos software, es patente que el software resultante dista mucho de ser confiable, y debemos asumir que el software que se produce no está libre de fallos. Dada la imposibilidad de diseñar o implementar sistemas libres de fallos, es necesario incorporar técnicas de mitigación de errores para mejorar la seguridad.

La presente tesis realiza aportaciones en tres de las principales técnicas de mitigación de errores de corrupción de memoria: *Stack Smashing Protector* (SSP), *Address Space Layout Randomisation* (ASLR) y *Automatic Software Diversification.*

SSP es una técnica de protección muy efectiva contra ataques de desbordamiento de buffer en pila, pero es sensible a ataques de fuerza bruta, en particular al peligroso ataque denominado *byte-for-byte.* Se ha propuesto una novedosa modificación del SSP, llamada RenewSSP, la cual elimina los ataques de fuerza bruta. Puede ser usada de manera completamente transparente con los programas existentes sin introducir sobrecarga. El RenewSSP es especialmente beneficioso en dos áreas de aplicación: Servidores de red (probado en Apache) y lanzadores de aplicaciones eficientes (probado en Android).

ASLR es un concepto genérico, del cual hay multitud de diseños e implementaciones. Se han analizado las dos implementaciones más relevantes de Linux (Vanilla Linux y PaX patch), encontrándose en ambas tanto debilidades como elementos mejorables. Teniendo en cuenta las mejoras tecnológicas en el soporte a la ejecución (compiladores y librerías), se ha propuesto un nuevo diseño del ASLR, llamado ASLR-NG, el cual: maximiza la entropía, soluciona el problema de la fragmentación y elimina las debilidades encontradas. Al igual que la solución propuesta para el SSP, la nueva propuesta de ASLR es transparente para las aplicaciones y compatible a nivel binario sin introducir sobrecarga. ASLR-NG ha sido implementado como un parche del núcleo de Linux para la versión 4.1.

La diversificación software es una técnica que cubre una amplia gama de fallos, incluidos los errores de memoria. La principal dificultad para aplicar esta técnica radica en la generación de las “variantes”, que son programas que tienen un comportamiento idéntico entre ellos ante entradas normales, pero tienen un comportamiento diferenciado en presencia de entradas anormales. Se ha propuesto una novedosa forma de generar variantes de forma automática a partir de un mismo código fuente, empleando la emulación de sistemas.

Una de las máximas de esta investigación ha sido la aplicabilidad de los resultados, por lo que se ha hecho especial hincapié en el desarrollo de prototipos sobre sistemas reales a la par que se llevaba a cabo el estudio teórico. Como resultado, las propuestas de esta tesis son directamente aplicables a sistemas reales, algunas de ellas ya están siendo explotadas en la práctica.

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Resum

Escola Tècnica Superior d'Enginyeria Informàtica
Departament d'Informàtica de Sistemes i Computadors

Doctor en Filosofia
(Enginyeria Informàtica)

Tècniques de ciberseguretat per mitigar l'explotació d'errors de memòria

per Héctor Marco Gisbert

La creació de programari suposa un dels reptes més complexos per al ser humà ja que requerix un alt grau d'abstracció. Encara que s'ha avançat molt en les metodologies per a la prevenció de les fallades de programari, és palès que el programari resultant dista molt de ser confiable, i hem d'assumir que el programari que es produïx no està lliure de fallades. Donada la impossibilitat de dissenyar o implementar sistemes lliures de fallades, és necessari incorporar tècniques de mitigació d'errors per a millorar la seguretat.

La present tesi realitza aportacions en tres de les principals tècniques de mitigació d'errors de corrupció de memòria: *Stack Smashing Protector* (SSP), *Address Space Layout Randomisation* (ASLR) i *Automatic Software Diversification*.

SSP és una tècnica de protecció molt efectiva contra atacs de desbordament de buffer en pila, però és sensible a atacs de força bruta, en particular al perillós atac denominat *byte-for-byte*.

S'ha proposat una nova modificació del SSP, RenewSSP, la qual elimina els atacs de força bruta. Pot ser usada de manera completament transparent amb els programes existents sense introduir sobrecàrrega. El RenewSSP és especialment beneficiós en dos àrees d'aplicació: servidors de xarxa (provat en Apache) i llançadors d'aplicacions eficients (provat en Android).

ASLR és un concepte genèric, del qual hi ha multitud de dissenys i implementacions. S'han analitzat les dos implementacions més rellevants de Linux (Vanilla Linux i PaX patch), trobant-se en ambdues tant debilitats com elements millorables. Tenint en compte les millores tecnològiques en el suport a l'execució (compiladors i llibreries), s'ha proposat un nou disseny de l'ASLR: ASLR-NG, el qual, maximitza l'entropia, soluciona el problema de la fragmentació i elimina les debilitats trobades. Igual que la solució proposada per al SSP, la nova proposta d'ASLR és transparent per a les aplicacions i compatible a nivell binari sense introduir sobrecàrrega. ASLR-NG ha sigut implementat com un pedaç del nucli de Linux per a la versió 4.1.

La diversificació de programari és una tècnica que cobrix una àmplia gamma de fallades, inclosos els errors de memòria. La principal dificultat per a aplicar esta tècnica radica en la generació de les “variants”, que són programes que tenen un comportament idèntic entre ells davant d'entrades normals, però tenen un comportament diferenciat en presència d'entrades anormals. S'ha proposat una nova forma de generar variants de forma automàtica a partir d'un mateix codi font, emprant l'emulació de sistemes.

Una de les màximes d'esta investigació ha sigut l'aplicabilitat dels resultats, per la qual cosa s'ha fet especial insistència en el desenvolupament de prototips sobre sistemes reals al mateix temps que es duia a terme l'estudi teòric. Com a resultat, les propostes d'esta tesi són directament aplicables a sistemes reals, algunes d'elles ja estan sent explotades en la pràctica.

Acknowledgements

I would like to thank my advisor, Ismael Ripoll, for giving me the chance to experience this exciting, four-year-long journey called a Ph.D. I will always remember your infinite patience and your unplayable advice. Thanks for putting light into my darkness.

I will not forget many of the people I have met over these past few years, some of whom are now good friends. They know to whom I am referring, thank you.

Special thanks go to my girlfriend for her constant encouragement. Without her this thesis could have not been possible.

Last but not the least, I extend my gratitude to my family, especially to my grandparents and my cousin, whom I think of as a brother – I thank all of you.

To all those who in one way or another always have believed in me, I will be eternally grateful to you.

thank you!

Contents

List of Figures

List of Tables

Acronyms

ABI	**A**pplication **B**inary **I**nterface
API	**A**pplication **P**rogramming **I**nterface
APT	**A**dvanced **P**ersistent **T**hreat
ARM	**A**dvanced **R**ISC **M**achine
ART	**A**ndroid **R**un**t**ime
ASLR	**A**ddress **S**pace **L**ayout **R**andomisation
AVI	**A**ttack $+$ **V**ulnerability $\rightarrow$ **I**ntrusion
CDF	**C**umulative **D**istribution **F**unction
CPU	**C**entral **P**rocessing **U**nit
CR	**C**arriage **R**eturn
CVE	**C**ommon **V**ulnerabilities and **E**xposures
CWE	**C**ommon **W**eakness **E**numeration
DEP	**D**ata **E**xecution **P**revention
DSA	**D**ynamic **S**torage/memory **A**llocator
DoS	**D**enial **o**f **S**ervice
EGLIBC	**E**mbedded **GLIBC**
ELF	**E**xecutable and **L**inkable **F**ormat
EMET	**E**nhanced **M**itigation **E**xperience **T**oolkit
FTP	**F**ile **T**ransfer **P**rotocol
GCC	**G**NU **C**ompiler **C**ollection
GNU	**GNU**'s Not Unix!
GOT	**G**lobal **O**ffset **T**able

GPL	**G**eneral **P**ublic **L**icense
GPU	**G**raphics **P**rocessing **U**nit
IP	**I**nstruction **P**ointer
ISR	**I**nstruction **S**et **R**andomisation
JNI	**J**ava **N**ative **I**nterface
JPEG	**J**oint **P**hotographic **E**xperts **G**roup
JRE	**J**ava **R**un-time **E**nvironment
LF	**L**ine **F**eed
LTS	**L**ong **T**erm **S**upport
MIPS	**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
MMU	**M**emory **M**anagement **U**nit
NX	**N**on-e**x**ecutable
OS	**O**perating **S**ystem
PAE	**P**hysical **A**ddress **E**xtension
PC	**P**rogram **C**ounter
PIC	**P**osition **I**ndependent **C**ode
PID	**P**rocess **ID**
PIE	**P**osition **I**ndependent **E**xecutable
PMF	**P**robability **M**ass **F**unction
PNG	**P**ortable **N**etwork **G**raphics
PPC	**P**ower**PC**
PRNG	**P**seudo **r**andom **N**umber **G**enerator
PoC	**P**roof **o**f **C**oncept
QEMU	**Q**uick **EMU**lator
RADIUS	**R**emote **A**uthentication **D**ial-**I**n **U**ser **S**ervice
RAF SSP	**R**e-new **A**fter **F**ork **S**tack **S**mashing **P**rotector
RAM	**R**andom-**A**ccess **M**emory
RELRO	**REL**ocation **R**ead-**O**nly

ROP	**R**eturn **O**riented **P**rogramming
RenewSSP	**Re**-new **S**tack **S**mashing **P**rotector
SANS	**S**ysAdmin **A**udit, **N**etworking and **S**ecurity **I**nstitute
SDK	**S**oftware **D**evelopment **K**it
SIS	**S**ingle **U**nix **S**pecification
SJLJ	**S**et**j**mp/**L**ong**j**mp
SPARC	**S**calable **P**rocessor **ARC**hitecture
SSH	**S**ecure **Sh**ell
SSPMD	**S**tack **S**mashing **P**rotector for **M**obile **D**evices
SSP	**S**tack **S**mashing **P**rotector
TLS	**T**hread **L**ocal **S**torage
UID	**U**nique **ID**entifier
USS	**U**nique **S**et **S**ize
VDSO	**V**irtual **D**ynamic **S**hared **O**bject
XOR	e**X**clusive **OR**

Symbols

∞	Infinite.
μ	Mean.
σ^2	Variance.
C	Number of entropy bits of the canary.
c	Number of values that the canary can take, $c = 2^C$.
k	Number of trials.
n	Number of random bytes of the canary.
R	Number of entropy bits of the ASLR.
r	Number of places where the ASLR can place a library/object, $r = 2^R$.

Chapter 1

Introduction

1.1 Motivation

Rapid advances in all aspects of today's information society, from technological support to economic models, have not been followed at the same pace by the cybersecurity field, and so there is a growing gap between the new solutions and functionality provided by software and hardware industry players and the accompanying new vulnerabilities and security issues created by them.

The more technological society becomes, the more exposed it is to cybersecurity problems. Recently, society passed the no-return point of technology dependency, i.e. our way of life would now not be the same without the internet and the ecosystem of smart devices and electronic services. Cyberspace is now a commodity that we need, in order to carry out most of our daily activities, both in cyberspace itself and in the real physical world.

All of the benefits and improvements that technology provides can be used against the society that uses it, if an attacker finds a way to do so, and as a result cyberspace is now viewed as the fourth battlefield, complete with new rules, new weapons and new actors. States are aware of this new scenario and the new challenges that will arise as a result; furthermore, this is not a hypothetical issue but a very real and urgent problem. Recently, Matt Blaze [1] warned the U.S. congress about this issue:

> *Unfortunately, modern computing and communications technologies, for all their benefits, are notoriously vulnerable to attack by criminals and hostile state actors. And just as the benefits of increased connectivity and more pervasive computing will continue to increase as technology advances, so too will the costs and risks we bear when this technology is maliciously compromised. It is a regrettable and yet time-tested paradox that our digital systems*

have largely become more vulnerable over time, even as almost every other aspect of the technology has, often wildly, improved.

Cybersecurity is a vast field of knowledge that covers all of the elements of the information society, such as risk analysis, espionage, critical infrastructure, international cyber-crime, cryptography, defensive measures, etc. This thesis focuses on the low-level technical aspects and mechanisms used to protect the system from malicious attacks.

Ultimately, an attacker needs to exploit a bug or weakness in a target system, in order to gain access to desired information or to gain the access level required to operate a device. The CWE [2], maintained by the MITRE, is a list of all known weaknesses and is continuously updated with new ways to bypass security or abuse computer systems. Nonetheless, despite the considerable amount of new weaknesses related to the internet and distributed systems, added in recent years to the CWE list, the classic and older "memory errors" are still a very dangerous and frequent family of errors. In fact, according to SANS [3], buffer overflow is ranked as the third most dangerous software error.

The objective of this thesis is to analyse and improve the defensive techniques used to mitigate software memory errors.

Memory error is a generic term which refers to a wide range of programming faults related to how a processor interprets the contents of the main memory and what happens when that memory is misused or misinterpreted by a program. It is important to differentiate the physical errors caused by the underlying hardware (memory cells), due to electrical or mechanical bugs, from the logical errors caused by incorrect coding. Essentially, the root cause of a memory error is a programming error, and it is therefore not related to hardware issues.

Despite the large amount of research effort expended on this subject, there is currently no satisfactory solution to the memory error problem [4]. Obviously, the current situation is far better than, say, one or two decades ago, because compilers are now able to analyse the code and detect, alert and even generate the correct code for certain types of memory error. Most of the new programming languages (Ada, Java, Python) try to hide the complexity of memory management all at once, by avoiding direct memory manipulation, which is an effective way to prevent most memory errors, albeit at a high cost in relation to expressiveness or overhead. Unfortunately, not all memory errors can be detected or captured before the code is released. The last line of defence is formed by a set of mitigation techniques that are active while the program is running, namely NX (Non-eXecutable), SSP (Stack Smashing Protector) and ASLR (Address Space Layout Randomisation), which do not remove the vulnerabilities per se but at least make them harder to exploit.

These three mitigation techniques are very effective and easy to implement, and so most systems (Windows, Linux, Mac, Android, etc.) include most of them or slightly modified versions thereof.

Most of the research effort focusing on the SSP and the ASLR techniques was done during the early 2000s. Once the technique was consolidated and proved to be effective, it was gradually introduced into products. Since then, no major improvements have been proposed – either technical innovations or better implementations –, since they are considered solid, stable and well-settled solutions.

After more than ten years, ever-growing computation capacity and programming models have advanced to the extent that the original requirements and conditions used to design these mitigation techniques are no longer valid, and so proposed solutions have to be revisited and upgraded.

Most of the contributions of this thesis are derived from a simple but powerful concept, namely diversification, which is the capacity of a system to change or modify the value of certain parameters or interfaces that are needed by an attacker to build a successful assault. It is important to differentiate between occultation and diversification. In the former, also known as 'security through obscurity', the design of the system is secret or unknown to the attacker, while in the latter case the design is widely known but the configuration or certain key data elements of the system are not known by the attacker, because they change over time. Security through obscurity is discouraged and not recommended [5].

Besides the classic process of the scientific method (observe, question, make a hypothesis, make a prediction, test the prediction and generalise), the following rules were added to the methodology for this thesis.

Practical results: I tried to avoid solutions which cannot be easily implemented on current systems; therefore, any solutions or improvements requiring important modifications to the application code, compilers, libraries or operating system were discarded. Also, solutions introducing excessive overheads (temporal or spatial) were also rejected.

It is typically easier to define new solutions if we focus mainly on the parameters and features that we wish to maximise (security, in our case) and dismiss or treat as secondary factors any drawbacks that may be involved, such as performance and complexity.

One of the premises was the urgency of the solutions. Although it is interesting to develop very advanced and efficient solutions, given the current situation in the cybersecurity field, ready-to-use solutions are required now, rather than perfect solutions tomorrow.

Real demonstrators: Besides the required formal validation and verification, the solutions presented herein were implemented and tested in real systems.

Real-world devices are complex, and sometimes there are interactions between elements that cannot be modelled properly, while certain interactions may even remain hidden until they are unintentionally triggered (as is the case with many security vulnerabilities). For example, the performance of current processors can be disturbed by many subtle and small changes, as seen in the Bulldozer cache aliasing problem [6].

Do not take for granted common assumptions: Since this thesis is about mature and well-studied techniques, it was very important to be as objective as possible and as distanced (i.e. not contaminated) as possible by previously published conclusions and solutions, since existing solutions, although very smart and efficient, are based on premises that were correct at the time when the conclusions were made, but they may no longer be valid or applicable to current systems.

Therefore, rather than starting from existing technology and building up from there, most of the work in this thesis consisted of questioning and challenging that technology.

Although there are small differences in the computational model and ABI (Application Binary Interface) programming between Linux, BSD, MacOS and Windows, the core ideas behind the protection techniques studied in this thesis are applicable to all of them. In order to avoid the burden of dealing with the implementation details of each system, this thesis only uses Linux as the development developing environment in which the new designs are tested and validated. The results obtained herein can be transferred to the other systems, with only minor technical customisations required.

1.2 Goals of the Thesis

The major goal of this thesis is to improve protection/mitigation techniques used to guard against memory errors. In particular we are concerned with the following issues:

Stack Smashing Protector: This covers a narrow range of memory errors, i.e. only stack buffer overflows, but nonetheless it is very effective against exploitation.

Unfortunately, SSP is prone to brute force attacks. Increases in network bandwidth and the processing power of current systems have reduced the time needed for these kinds of attack to be effective, and there is an especially dangerous type of brute force attack called 'byte-for-byte', which can bypass protection protocols in just a very few number of trials, regardless of the processor's word width (32 or 64 bits).

Address Space Layout Randomisation: This is a generic mechanism which is difficult to exploit. Rather than providing protection against a specific class of faults, the ASLR works as an additional problem that must be solved by the attacker in order to execute a remote code, typically via ROP [7] programming or ret2x [8].

Depending on how the ASLR has been designed and implemented, if the attacker is able to acquire the address (via an info leak) of the application, then most of the memory layout can be de-randomised – in which case the ASLR can be considered defeated. This problem has been shown in the chapter 6, but there are also other design problems that also deserve better solutions.

Automatic SW Diversification: Software diversification has been used as a mechanism to detect and recover from a large variety of software errors (not only memory errors). It is viewed as the software version of the well-known TRM (triple modular redundancy) [9] fault-tolerant mechanism.

Conversely to hardware implementation, redundancy cannot be obtained by just making an exact copy of the software, as is done when then the technique is used in electronic systems, because all of them will fail in exactly the same way. What is therefore required is to have two or more 'versions' of the same program, all of which respond similarly when there are no errors but which have a differentiated output (manifests differently) in the case of an error.

How to generate strong diversification efficiently is an open issue that is addressed in the thesis.

1.3 Contributions of the Thesis

The main contributions of this thesis are summarised as follows:

1.3.1 RenewSSP: Renew Stack Smashing Protector

1. A **statistical analysis** of the effectiveness of SSP, NX and ASLR against local and remote attacks considers the combined protection of the techniques as well as their individual effectiveness.

2. An improvement is made to the SSP, called **renewSSP**, which prevents brute force attacks against the SSP technique itself. As a result, the SSP is now much more effective. The technique can be used transparently on production systems (by pre-loading a shared library) and has no appreciable overheads.

   A software patent has been requested to cover this new technology.

3. The software architecture of the Android OS relies on the Zygote process, which acts as a fast application launcher. Unfortunately, this solution has important security drawbacks. The **renewSSP technique** has been shown to be an effective solution for restoring the security of the SSP on Android.

1.3.2 ASLR-NG: Address Space Layout Randomisation Next Generation

1. A new kind of weakness, called *offset2lib*, has been described.

2. A proof of concept for offset2lib weakness has been developed and ranked as **1-day** vulnerability (by PacketStorm Security).

3. The classic process memory layout model has been revisited, questioned and reworked, thus allowing for improvements to be made to the current ASLR design.

4. A new version of the ASLR, called *ASLR-NG*, has been designed with the following features:

   - It does not suffer from offset2lib vulnerability.
   - It defines several new forms (or dimensions) of entropy.
   - It can use the full memory space entropy, which provides maximum randomness (and thereby maximum protection).
   - It is binary-compatible with existing applications; in other words, existing software does not need to be recompiled or modified in order to gain benefits.

5. The new version has been implemented on the Linux kernel.

1.3.3 DRITAE: Automatic SW Diversification

A new diversification architecture, called DRITAE, which has the following attributes:

1. Processor cross-compilation has been identified as a new mechanism to produce binary diversification automatically: variants.
2. User mode emulation has been identified as a mechanism that can be employed to run different variants on a single system. This way, most of the state of the application can be maintained by the operating system in a very efficient way.
3. A novel recovery strategy for network servers against attempts to exploit zero-day bugs.
4. A proof of concept monitor has been implemented which implements the recovery strategy.

1.3.4 Other Contributions

During this thesis' analysis of the technological state of the art (the Linux kernel and other system software packages), several bugs and weaknesses were found and reported to the authors. Also, these bugs were submitted to MITRE for consideration as CVEs. A total of **seven CVEs** (common vulnerabilities and exposures) were finally assigned, most of which are directly linked with the main research line of this thesis (application buffer overflows, kernel bugs and missing glibc features), though there are other CVEs related to privilege escalation or DoS.

1.4 Thesis Outline

This thesis is structured in three thematic parts, and each part is organised in chapters which correspond to already published works or internal reports that will be published shortly.

Part I is about the current state of the stack smashing protector technique, any weaknesses that threaten the protection technique itself and the new **renewSSP** solution proposed and implemented to toughen the SSP. Part II covers limitations of the ASLR design, such as **Offset2lib**, and proposes a new design jointly with the new implementation named **ASLR-NG**. Part III is about automatic software diversification and the new **DRITAE** architecture, while the final part, Part IV, summarises and concludes and then outlines future research work.

Part I

Stack Smashing Protector (SSP)

Chapter 2

Preventing Brute Force Attacks against the SSP

This chapter introduces the conditions that must be met when a program renews the reference canary on a child process when a new process is forked. The technique is referred to as 'RAF' (renew after fork) SSP.

The technique is generalised and ported to other systems in later chapters.

Contents

2.1 Introduction

A decade ago, buffer overflows, especially stack-smashing, was the most dangerous threat to computer system security. Over the last few years, several techniques have been developed to mitigate the ability to exploit this kind of programming fault [10, 11]. Stack-smashing protector (SSP), address space layout randomisation (ASLR) and Non-eXecutable stack (NX) are widely used in most systems, due to their low overheads, simplicity and effectiveness.

Following the classic measure/counter-measure sequence, a few years after the introduction of each protection technique, a method to bypass or reduce their effectiveness was introduced. The SSP can be bypassed using brute force or by overwriting non-shielded data [12, 13], the ASLR can be bypassed using brute force attacks [14] and the NX, which effectively blocks the execution of injected code, can be bypassed using ROP (return-oriented programming)[15]. In spite of many existing counter-measures, these techniques are still effective protection methods, and in some cases they are the only barrier against attacks, until software is upgraded to remove a specific vulnerability.

Unfortunately, the forked and pre-forked networking server architectures are particularly prone to brute force attacks, as all children processes inherit/share the same memory layout and the same canary as the parent process. An attacker can try – in bounded time – all the possible values of a canary (for SSP) and memory layouts (for ASLR) until the correct ones are found. There is a very dangerous form of SSP vulnerability, called byte-for-byte, which allows an attacker to try each byte of the canary independently, which then allows one to find the value of the canary by carrying out just a few hundred trials (the system is defeated in a matter of seconds).

We present a modification of the SSP technique which consists of setting a new random value of the canary for each child process when the `fork()` system call is invoked. The technique is called 'RAF SSP' (re-new after fork SSP).

Re-randomising the canary has not been seriously considered [13], mainly due to two factors: first, protection increases only by a factor of two, on average, when compared with the standard SSP on a system with no other protection techniques. And second, the complexity of the implementation would not be worth the protection improvement.

The RAF SSP technique greatly increases (by several orders of magnitude) the difficulty of an attack when the three protection techniques (SSP+ASLR+NX) are employed, as is the case in most systems. Regarding the problems that may cause a canary change on a running process, we have identified that the error confinement that represents each forked thread is also a de facto stack confinement which allows one to change the value of the reference canary with no impact on the correct operation of a process.

2.1.1 Benefits of our proposal

The main properties of the RAF SSP are:

- It can be used by just pre-loading a shared library. There is no need to modify the source code of the networking servers, nor recompile the application, nor modify system libraries or the compiler.
- It **prevents brute force attacks** against canary stack protection mechanisms.
- The SSP byte-for-byte attack is no longer applicable to the RAF SSP.
- It multiplies the effectiveness of the combined protection of the SSP, ASLR and NX techniques. On a standard 32-bit system the cost of breaking a system **takes 512 times more trials** on average (from 2^{23} trials to 2^{32} on a 32-bit system).
- The overhead introduced is negligible.

The RAF SSP is especially useful for networked servers, but it is not limited to them. We tested it on a complete Linux distribution by modifying the standard C library, with full functionality and no appreciable performance penalty.

2.2 Background & assumptions

2.2.1 Network server architectures

Network server software architectures have been extensively studied and analysed due to the importance of Web servers in the current network infrastructure. Attending to the processing model, we will focus on two basic models, paying special attention to the robustness of each approach. A complete list of the models is beyond the scope of this paper (see [16]).

Multi-thread: each connection is handled on a dedicated thread. Multiple clients can be attended to concurrently with a low overhead. The main drawback is that a crash in any thread may kill the whole server or put the server into an inconsistent state. There is a single error confinement region, namely the server process.

Multi-process: each connection is handled by a separate (child) process. There are two variants. First is the *forking server*, where a child process is created explicitly to serve each request. As soon as the client request is finished, the child process exists. The server process is able to attend to client requests at any time, except while the child is being created (while forking). The other variant is the *pre-forking server*. Several sub-processes are forked before any connections are handled (when the server starts). Each child blocks a new connection, handles the connection and then waits for the next connection. This removes the overhead of the `fork()` call at the time of accepting a new connection. The crashing of a child process has a limited impact on the operation of the server, and only the client that caused/suffered the crash has a failure. The error confinement region for this mode is isolated for each process.

A good compromise between performance and robustness (error confinement) is the multi-process, multi-threaded hybrid used in Apache.

2.2.2 Stack-smashing protection (SSP)

The first proposal was presented in [17] and then improved over the years. Without loss of generality, we will assume that the stack grows downwards, i.e. to lower addresses. We will use the x86 architecture in our examples.

The SSP technique [18] is a compiler extension which adds a guard (the canary) between the protected region of the stack and local buffers. Originally, the canary was placed right after the return address, since it was the target of most attacks. Over the years, new attack strategies have been developed (see section 2.3) which have encouraged some enhancements [19]. As of GCC v4.6.3, the stack-smashing protector consists of the following:

- Both the return address and the saved stack frame pointer are guarded by the frame canary.
- Local variables are reordered so that buffers are located first (higher addresses), and then below them sit the scalar variables and the saved registers. This way, buffer overflows (which typically grow upwards) will not overwrite scalar variables.

Figure 2.1 sketches the layout of a stack with two function frames. The reference canary is represented as a small bird on the right-hand side, and the frame canary is the bird on each frame. The compiler emits extra code

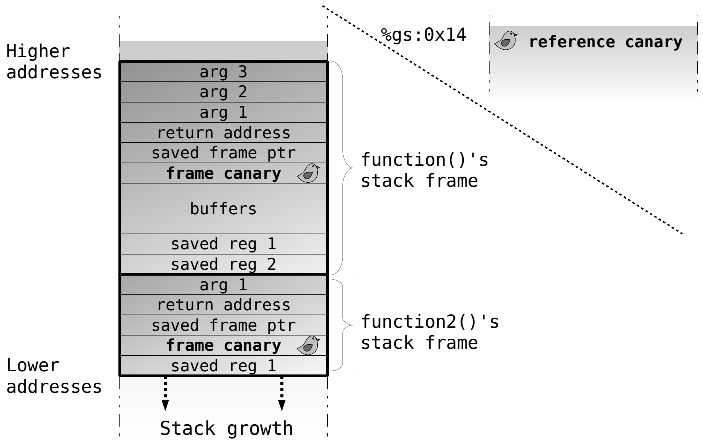


FIGURE 2.1: x86 Stack layout.

in the prologue and epilogue of each protected function, in order to initialise and check the value of the canary.

The value of the canary is chosen such that it prevents, where possible, the effective exploitation of a buffer overflow and detects the occurrence of an overflow. Attending to these issues, two (XOR canaries are not included, because they have more overheads than the other types and the same properties as random canaries) kinds of canary values have been proposed:

- Terminator value: this value is composed of different string terminators (CR, LF, NULL and -1).
- Random value: this value is a random value selected during the process initialisation phase. The attacker needs to know the actual value in order to build the attack. As long as the value remains secret, the attack will be prevented.

In most implementations, the canary value is a word with all of it bytes randomised, except one that is zeroed.

Since the value of the canary is not a constant but a random value chosen when the program starts, this value has to be stored somewhere in the program memory (or in a dedicated processor register, if available). In the x86 and x86-64 architectures the reference canary is stored in a special data segment which is not accessible as a 'normal' variable and cannot be overwritten or read.

The numerical examples provided in this paper refer to a GNU/Linux x86 architecture.

2.3 Threats

In this section, rather than a detailed explanation on how to bypass the SSP, we will present only the weaknesses of the stack canary that enable the possibility of an attack. In [20], the author explains the process employed to remotely exploit a buffer overflow on systems equipped with these techniques.

Basically, there are two ways to bypass the canary:

1. Overwriting the target data (return address, function pointer, etc.) without needing to overwrite the frame canary.
2. Overwriting the frame canary with the correct value.

Obviously, data that are not guarded by the canary (exception handlers, function pointers in data structures, etc.) are prone to other forms of buffer overflow, but since our technique does not increase the coverage (detection capability) of the basic canary technique but reinforces the protection of the already protected items, we will focus on brute force attacks against the canary value.

The second way to bypass the canary requires an attacker knowing the actual value of the canary. Processes created with `fork()` are duplicates of the calling process. Both parent and child have the same canary value. On a forked server, where the service is attended to by children of the server process, an attacker can build **brute force attacks** by guessing the value of the canary as many times as needed.

Depending on the granularity of how the attacker can flood the buffer (word or byte overflow), there are two different brute force attacks: *full brute force* and *byte-for-byte*.

2.3.1 Full brute force attack

The frame canary word is overwritten on each trial. If the guessed word is not correct, the child process detects the error and aborts. As a consequence, the attacker does not receive a reply, which is interpreted as an incorrect guess. The guessed value is discarded, and the attacker then proceeds with another value until all the possible values have been guessed.

On most 32-bit systems the canary (word) has 3 random bytes plus one zeroed. In the worst case, the number of trials is 2^{24}, and $2^{23} = 8,388,608$ on average. These figures may deter a remote attacker but not a local attacker, which may still break the system in just a few hours.

2.3.2 Byte-for-byte brute force attack

If the attackers have fine-grained control over the number of bytes that overflow, then a byte-for-byte attack can be constructed. The attack consists of overwriting only the first byte of the canary until the child does not crash. All the values from 0 to 255 are tested sequentially until successful. The last byte tested is the first byte of the canary. The remaining bytes of the canary are obtained following the same strategy.

This kind of attack is very dangerous, because a system can be broken after only $3 \times 256 = 768$ trials. For this reason, most canary implementations set to zero one of the canary bytes (the most significant in x86) in order to prevent 'byte for byte' attacks when the overflow is performed by string copy functions.

2.4 Proposed strategy

The following section first describes the strategy idea. Below, is an example of a stack evolution which is described in order to clarify the strategy. Finally a brief of some special cases where the canary reference should be changed carefully.

2.4.1 Observations

Our proposal is based on the following observations:

Observation 1. For most applications, especially networking servers, after a `fork()` operation the child process executes a flow of code which ends with an explicit call to the `exit()` system call. In other words, the child process does not return from the function that started the child code.

Even those applications that do not end with an `exit()` after a `fork()`, they do not suppose a problem because the canary is not checked when returning to the parent functions in most cases.

We validated this observation by both 1) analysing the code of several servers and 2) empirically, by running a complete GNU/Linux distribution where processes that return after a fork are killed.

Observation 2. Each child process in a network server defines an error confinement region. That is, any error that occurs in relation in a child will not affect the correct operation of the parent and sibling processes.

Observation 3. There is a single reference canary per process which is stored in a protected area and initialised during process start-up. It is copied in each stack frame between the saved stack frame and the buffers.

Observation 4. Integrity (comparing the frame canary against the reference canary) is only checked at the end of each function, right before the returning instruction.

Observation 5. Only the value of the frame canary of the current stack is checked against the reference canary.

2.4.2 Renew canary at fork (RAF SSP) strategy

The renew canary at fork (RAF SSP) strategy involves renewing the value of the reference canary of the child process right after it has been created (forked). The new value is also a random value. Every child process has a different reference canary.

From our observations 1 we know that the child code will not return ,and so the stacked frame canaries will never be tested 5. Therefore, they do not need to be updated.

Although it is not difficult to check whether or not a function ever returns, most compilers provide the `noreturn` function attribute, which declares a function as non-returning. The compiler generates more efficient code and checks (at compile time) whether or not the function honours the desired behaviour. It is advisable that this function, i.e. where the canary is renewed, has this attribute.

When the attacker guesses an incorrect value, the child is killed by the stack protector detection mechanism and a new child with a new canary is started. As a result, **brute force attacks cannot be built**.

2.4.3 Illustrative examples

Example 1: Figure 2.2 represents the evolution of the code on listing 2.1. Different canary values are represented by different colours. When a function is called, the stack frame is set up, copying the reference canary into the frame canary (see stack state at time 1). Upon return, the frame canary is compared with the reference canary, which is represented by a black diamond with an equal sign inside. The `renew_canary()` operation changes only the reference canary, the stack is not modified (from time 3 to 4) and it defines a point of no-return. The current function (`foo()` at time 4) cannot return, since there will be a mismatch between the reference and the frame canaries.

Note that the frame canary of the previous stacked frames keeps the old reference canary value while the new frames have the new canary (times 5, 6 and 7). As long as the process never returns from those functions, it will

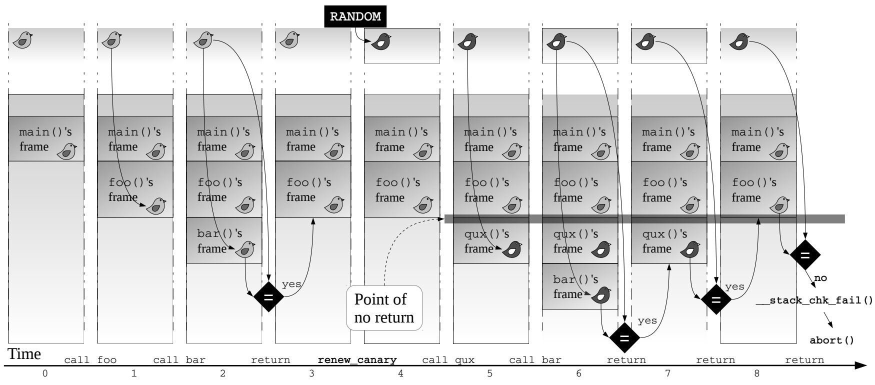


FIGURE 2.2: Stack evolution program changing the reference canary.

```
void foo(){                |  void bar(){
  bar();                   |    return;
  renew_canary();          |  }
  qux();                   |
  return; //Fails          |
}                          |  void qux(){
                           |    bar();
int main(){                |    return;
  foo();                   |  }
  return 0;                |
}                          |
```

LISTING 2.1: Code example which
renew the reference canary.

not be a problem. Also, note that the same function (`bar()` on times 2 and 6) has different frame canaries when called with different reference canaries and they return correctly in both cases.

If the function that changed the reference canary returns (time 8), there will a canary mismatch, which in turn will result in a process abort.

Example 2: Figure 2.3 shows the state of the stack and the reference canary on a forking server using the RAF SSP technique. All of the stack frames used by a child have the same frame canary value, which is different from the parent and from other children. Since each child process defines an error confinement region, our strategy randomises the canary in each confinement region.

In the case of a forking server, each client is attended to by a different child process. Since the child always terminates after attending to the client,

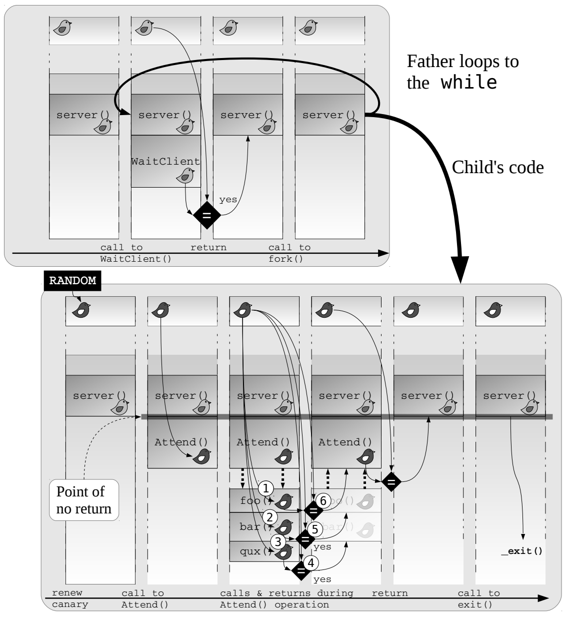


FIGURE 2.3: Stack evolution of the code on listing 2.2.

```
void server(){               |    void Attend(){
  ...                        |      foo();
  while(1) {                 |      ....
    client=WaitClient();     |    }
    if (fork()==0) {         |    int foo(){ bar();
    }
     renew_canary();         |
     Attend();               |    int bar(){ qux();
    }
     _exit();                |
  }                          |    char qux(){};
}  ...
```

LISTING 2.2: Basic forking server example.

the canary is renewed on every connexion, regardless of whether or not it is requested by a legal client or an attacker. On a pre-forked architecture, a child is running and attending requests until it is killed (by the main server process, to reduce the number of active processes, or because it has crashed as a result of an attack).

Taking into account observation 4, the frame canary of a non-returning function is never checked against the reference canary. Therefore, a non-returning function can safely change the value of the reference canary. All subsequent function calls (invoked from the non-returning function) will use the new reference canary for building the stack frame. Upon returning, the canary checks will match, because the reference canary and the frame canary will be the same (assuming the stack has not been smashed).

2.4.4 Special considerations

The `setjmp()/longjmp()` library provides a control flow facility that breaks the 'normal' control flow of functions, namely a call and return sequence. The `setjmp()` saves the stack context/environment in a variable that can later be restored by the `longjmp()` function. This backward jump does not involve any return operation and does not check the integrity of the frame canaries, and so as a result it can be used safely. Nonetheless, if the reference canary has been renewed between the time when the `setjmp()` was called and when the `longjmp()` is invoked, then there will be a mismatch between the reference canary and the stack-canary for the function where the `setjmp()` was performed.

This special case can be addressed in two different ways:

- Do not return from functions that call `setjmp()`. This behaviour can be forced by declaring that functions obey a `noreturn` command. Note that the function that uses `setjmp()` can call up nested functions at any time, but it cannot return.
- A more robust and transparent solution consists of modifying the struct where `setjmp()` saves the context/environment, by adding a new field to store the value of the reference canary at the time the `setjmp()` is called. Later, the `longjmp()` will restore the reference canary value using the one that is being stored. With this solution there is no limitation to control flow.

2.5 Implementation

Proof of concept for the proposal has been implemented as a shared library which overrides the `fork()` call. The library is called `libraf.so`, and the code for the library is in listing 2.3.

The `fork()` function, on listing 2.3, calls the native `fork()` function, following which the reference canary of the child is renewed by calling the `renew_rnd_stack_chk_guard()` function, which is basically a copy of the library code used to set-up the canary. All but one bytes are random values read from /dev/urandom, which is the same source of randomness as used by the standard canary.

```
#ifdef __i386__
# define THREAD_SET_STACK_GUARD(x) \
   asm ("mov %0, %%gs:0x14" ::"r" (x) : "memory");
#elif defined __x86_64__
# define THREAD_SET_STACK_GUARD(x) \
   asm ("mov %0, %%fs:0x28" ::"r" (x) : "memory");
#endif
pid_t (*native_fork) (void);
static void __raf_fork_init(void) {
   native_fork = dlsym(RTLD_NEXT, "fork");
   if (NULL == native_fork) {
      fprintf(stderr, "Error in `dlsym`: %s\n",
    dlerror());
   }
}
static void renew_rnd_stack_chk_guard(void) {
   union {
      uintptr_t num;
      unsigned char bytes[sizeof (uintptr_t)];
   } ret;
   const size_t ranb = sizeof(ret.bytes) - 1;
   ret.num = 0;
   int fd = __open("/dev/urandom", O_RDONLY);
   if (fd >= 0) {
      if (__read(fd, ret.bytes + 1, ranb) == ranb){
         THREAD_SET_STACK_GUARD(ret.num);
      }
      __close (fd);
   }
}
pid_t fork(void) {
    pid_t pid;
    if (native_fork==NULL) __raf_fork_init();
    pid = native_fork();
    if (pid == 0) renew_rnd_stack_chk_guard();
    return pid;
}
```

LISTING 2.3: RAF SSP implemented as shared library: libraf.so.

Although the compiler will not add stack protector code to any of its functions, because they do not have local buffers (larger than 8 bytes), it is advisable to compile the shared library with the option -fno-stack-protector, in order to be sure that the compiler does not add it; otherwise, the canary renewal will be detected as a stack corruption and the program will be aborted.

In order to use the new version of `fork()`, the server has to be launched with the LD_PRELOAD=`libraf.so` as follows:

```
$ LD_PRELOAD=libraf.so apachectl start
```

Another way to include the proposal in a running system is by modifying the standard C library. We tested the proposal by modifying the code of the `fork()` function of the GNU eglibc. Nevertheless, for brevity, we describe it here as a new library call, namely `raf_fork()`. The modification basically consists of calling the canary set-up code `_dl_setup_stack_chk_guard()` again, right after calling the fork.

```
void renew_canary(void) {
  /* Renew the stack checker's canary.  */
  uintptr_t stack_chk_guard =
    _dl_setup_stack_chk_guard (NULL);
#ifdef THREAD_SET_STACK_GUARD
  THREAD_SET_STACK_GUARD (stack_chk_guard);
#else
  __stack_chk_guard = stack_chk_guard;
#endif
}
pid_t raf_fork (void) {
   if (fork()==0) renew_canary();
   return pid;
}
```

Listing 2.4: Implemented as a new service: `raf_fork()`

The modification of the `setjmp/longjmp` family involves increasing the size of the `jmp_buf` array, to store the value of the current reference canary at the `setjmp` (which is done with just one assignment instruction), and then to restore it on the `longjmp` function.

2.6 Statistical evaluation

We analyse the protection provided by RAF SSP as both a stand-alone technique and when combined with the ASLR and NX techniques. The cost is measured as the number of attempts (trials) needed by the attackers to bypass them. Since the NX technique prevents remote code injection, in order to exploit stack smash vulnerability it is necessary to bypass the canary plus the ASLR all at once, as presented in section 2.6.2. The unrealistic attack on the canary as a stand-alone technique has been included for completeness.

Let k be the number of trials until the system is defeated, let c be the number of different values that can take the canary and let r be the number of different positions where the ASLR can place the code.

The system is broken when the secrets are correctly guessed. We are interested in the probability distribution of the process defined as "*the probability that the first success requires k number of trials*". Larger values of k are good for defenders and bad for attackers.

The plots shown in Figures 2.4 and 2.5 are for a standard Ubuntu 32-bit system, where the canary has 3 bytes (which gives a range of $c = 2^{24}$ values) and the ASLR has 8 bits (256 values) of entropy. Note that the 256 values of the ASLR entropy represent the worst case for the attackers, assuming that only a mapped library (typically the libc) it is enough to build the attack.

2.6.1 Bypassing only the canary

As described in section 2.3, there are two main ways to attack the canary: full search and byte-for-byte.

Full search attack: Statistically, a brute force attack is described as "**sampling without replacement**," and since all the values have the same probability ($\frac{1}{c}$) it is modelled by uniform distribution with a support range of $[1, c]$ and a mean of $\frac{c+1}{2}$.

When the RAF SSP is used, the attacker cannot launch a brute force attack, because incorrect values cannot be discarded. The only strategy for an attacker is to select a valid canary value (the value of which does not matter) and keep trying the same value until the canary matches with the one randomly chosen by the server. The attacker can change the value of the tried canary, but it does not improve the chances of finding a match. This trial process is described as "**sampling with replacement**" and is modelled by a geometric distribution with a support range of $[1, \infty[$ and a mean of c.

Standard SSP requires at most 2^{24} trials[1] to be sure that the canary value is found, but with the RAF SSP it is impossible to cover all cases. On average, the RAF SSP requires only twice as many trials to be broken. Three times the mean is needed to have a 95% chance of breaking in.

Byte-for-byte attack: On a byte-for-byte attack, the process of finding each byte is modelled as a uniform distribution whose mean is 256/2 and support range is $[1, 256]$. The attack on the 3 (for 32-bit systems) or 7 bytes (64-bit systems) is modelled as the sum of 3 or 7 uniform random variables,

[1] On a 32-bit Ubuntu.

respectively. Using the central limit theorem, the resulting distribution can be approximated to a normal distribution with $\mu = 256n/2$, where n is the number of random bytes of the canary ($n = log_2(c)/8$), with support of $[1, 256n]$.

When the RAF SSP is in situ, a byte-for-byte attack cannot be employed, because any incorrect guess will trigger a renewal of the full canary (all the bytes), which invalidates any previous correctly guessed byte. Therefore, the attacker is forced to use sampling with replacement against the full canary. Figure 2.4 compares the cumulative distribution function (CDF) of the attack to a 'byte for byte' exploitable overflow using the standard SSP and the proposed RAF SSP. The x-axis is logarithmic.

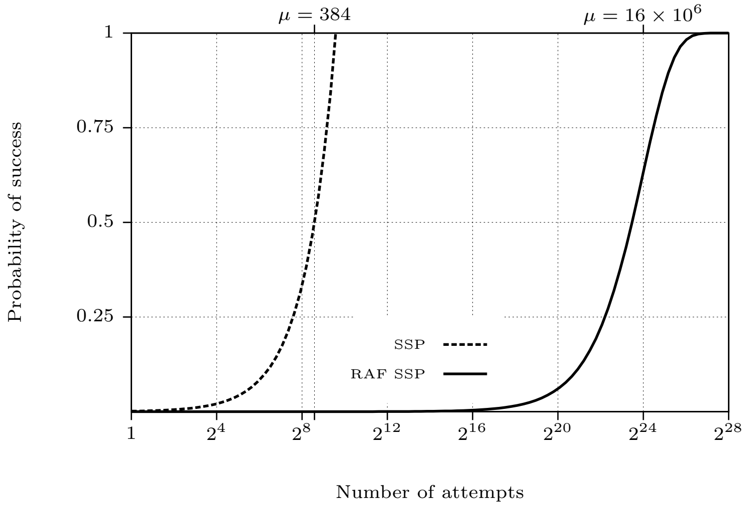


FIGURE 2.4: Byte-for-byte vulnerability.

With standard SSP, the attacker needs at most 768 trials to break the system (and 384 on average). With this figure, the standard canary technique provides weak protection for this kind of bug. On the other hand, the RAF SSP disables the ability to split the attack into bytes, and so the same attacker requires $2^{24} = 16 \times 10^6$ trials on average to break it (it is a geometric distribution with $\mu = c$), which represents an **improvement of five orders of magnitude**.

2.6.2 Bypassing SSP + ASLR + NX

In a real system, all of these three complementary techniques are used simultaneously. Therefore, in order to exploit the fault, the attackers must:

1. Bypass NX protection. NX prevents the execution of injected code, and attackers are forced to 're-use' the code already present in the process by means of ROP programming.
2. Bypass SSP protection, which involves finding the actual value of the canary.
3. Bypass ASLR protection, which requires knowing the absolute address of the attacker's code, i.e. the entry point of the ROP sequence.

The attackers have to prepare/program the ROP sequence off-line. If we suppose that they know the code of the server, then only the address of the entry point to the ROP sequence is unknown, and so the rest of the ROP sequence is relative to that entry point.

It is important to note that both the ASLR and the canary techniques have the same weakness on forking servers: all the children have the same memory layout as well as the same canary value. Unfortunately, as far as the authors know, there is not a simple method to re-randomise the ASLR on forked children. We shall therefore assume that all the forked children

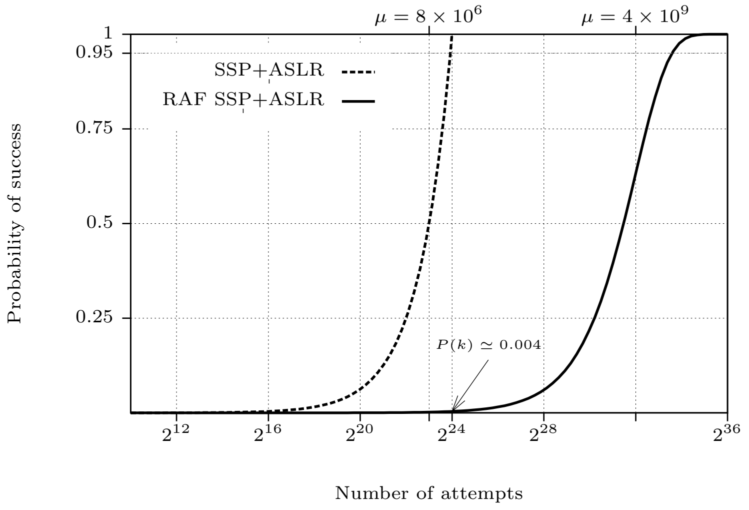


FIGURE 2.5: Success of an attack against both SSP and ALSR.

	Standard SSP		RAF SSP		
	Trials to break in 100%	Mean	Trials to break in 100%	Mean	Mean increased by a factor of
SSP_bfb	3×2^8	3×2^7	∞	2^{24}	$43,691$
SSP_full	2^{24}	2^{23}	∞	2^{24}	2
SSP_bfb+ASLR	$3 \times 2^8 + 2^8$	2^9	∞	2^{32}	$8,388,608$
SSP_full+ALSR	$2^{24} + 2^8$	$2^{23} + 2^7$	∞	2^{32}	512

Table 2.1: Standard SSP versus RAF SSP ($c = 2^{24}, r = 2^8$).

have the same ASLR value which allows the attacker to perform a brute force assault. The attacker has to create a brute force (or a probabilistic) attack to obtain the value of the two secrets: i) **canary value** and ii) **ROP entry address**. Let r be the number of different values, where the ASLR algorithm can map memory (i.e. the ASLR entropy).

We assume that the return address can only be overwritten if the canary is known. That is, the server's code always checks the canary first, and it only returns from the function if it is correct. This behaviour allows the attacker to split the attack into two phases: first the value of the canary is found and then the value of the ROP entry address. A detailed analysis of how the three techniques can be bypassed can be found in [20]. The attack on the ASLR follows the same pattern as the attack on the canary – it is a uniform distribution, sampling without replacement, whose mean is $r/2$ and support is $[1, r]$.

Using standard SSP, the attack on the server is modelled as the sum of two uniform distributions (which gives a trapezoidal distribution). If one of the uniforms has a much larger support range than the other ($c \gg r$), as in our case, the sum can be approximated by a simple uniform distribution with a mean of $(c + r)/2$ and a range of $[2, c + r - 1]$.

When the RAF SSP is used, it is not possible to split the attack, because any incorrect guess (either a canary value or an ROP entry point) causes a canary renewal[2]. Since $c \gg r$, it is possible to approximate the resulting distribution to a geometric with a mean of $c \times r$ and support $[2, \infty[$.

Figure 2.5 shows the success probability of an attack on a standard system and when the RAF SSP technique is used on a real system. The x-axis is logarithmic. When the RAF SSP is used, the cost of the attack is calculated as the result of **multiplying** the cost of each part – the canary and the ASLR, while in a normal system it is only the **sum** of each part. It takes

[2]A detailed analysis of attacker strategies and associated statistical distributions is beyond the scope of this paper. It will be published in an upcoming paper.

at most $2^{24} + 2^{8}$ trials to break into a standard system, and the system is destroyed with a 100% certainty. Using the RAF SSP technique, the chances of breaking into the system using the same number of trials is only 0.004%. On average, as table 2.1 shows, a 32-bit system using the RAF SSP **requires** 512 **times more trials** to break.

2.7 Experimental evaluation

The RAF SSP relies on the infrastructure of the standard SSP. Therefore, it does not increase the runtime cost operation of the application except when the fork operation is invoked. The temporal overhead can be reduced to the cost of generating a random word every time the `fork()` system call is made.

The **temporal cost** of renewing the reference canary is determined by the *r*enew_randomly_stack_chk_guard() function. The average cost of calling this function one million times is $2\mu s$ on an Intel Core™ 2 Duo CPU running at 2.4Ghz.

Regarding **spatial cost**, the memory size of the application does not increase, no new data structures are used and no new code is generated by the compiler. In addition, no new data structures or buffers are needed in the library, and only a few lines of code have to be included.

The RAF SSP was tested with the following network servers: apache2, lighttpd, proftpd and samba. For the sake of brevity, we will describe herein only the apache2 benchmarks.

We used the Apache (apache2-mpm-prefork) binary included in the Ubuntu (12.04) distribution with the default configurations (listing 2.5) and the *Apache HTTP server benchmarking tool (ab)* tool to generate the client workload.

```
<IfModule mpm_prefork_module>
    StartServers          5
    MinSpareServers       5
    MaxSpareServers      10
    MaxClients          150
    MaxRequestsPerChild   0
</IfModule>
```

Listing 2.5: Apache configuration parameters.

The ab tool was configured to perform requests 1KB, 10KB and 100KB in size, and each size was tested with 10, 50 and 100 concurrent requests (concurrency column). Each experiment consisted of 10^{6} requests.

		Latency (ms)		**Throughput (KB/s)**	
	Concurrency	SSP	RAF SSP	SSP	RAF SSP
1KB	10	0.094	0.093	12	12
	50	0.097	0.097	12	12
	100	0.097	0.098	12	12
10KB	10	0.095	0.095	104	104
	50	0.099	0.099	101	100
	100	0.102	0.101	98	99
100KB	10	0.135	0.135	725	723
	50	0.142	0.143	690	683
	100	0.164	0.164	598	596

Table 2.2: Performance overhead comparative.

Table 2.2 shows the average of each experiment. There are no significant differences between standard and RAF SSP apart from the variability introduced by the processor and operating system features. The small overhead caused by RAF SSP is practically undetectable when analysing the complete operation of the server.

We installed a modified version of the eglibc (with the RAF SSP enabled at the fork) in a Ubuntu Linux distribution, and all the tested applications worked correctly: all graphical services, several browsers, several text editors (LibreOffice), Java Open JDK interpreter, etc.

2.8 Discussion

The sequence `fork() + exec()` provides a very robust security schema because of strong decoupling between the parent and the children. The exec call renews both the value of the canary and the addresses of the code (ASLR). With respect to the canary, our technique is as powerful as calling exec, albeit without the high overheads of the exec call.

Contrary to other approaches, where the canary is different in each stack frame[21], our approach changes the canary for every error confinement region. From the attacker's point of view, the target stack frame is the one that belongs to the faulty function; the rest of the stack frames are of no interest to the attacker. On a network server, only the code executed by the child sever is prone to attacks, and in that code the canary is always re-randomised by the RAF SSP.

Our technique does not modify the coverage protection of the SSP technique. If a programming error can be exploited in such a way that the SSP technique fails to detect it, then RAF SSP will fail, too. Nevertheless, if stack-smashing is detected, then our modified system will also detect it and make it more difficult to exploit.

Since the overhead of the RAF SSP occurs only at the fork call, the fewer the forks done by the server, the lower the overhead. For example, a pre-forked server has zero overheads as long as no new children are launched. If the server's children processes are being killed during an attack (due to incorrect guesses regarding the brute force attack), then any new children come fully equipped with newly randomised canaries. That is, the RAF SSP acts only when it is effectively required to do so, while during normal operations it has zero impact.

In the section 2.6.2 we evaluate the cost of an attack when the three most common protection techniques are simultaneously employed, and we show that combined effectiveness can be computed as the product of each one because the attacks cannot be split. Since the canary is typically the first barrier to be bypassed, the RAF SSP causes the same multiplicative effect when combined with other techniques, as long as any incorrect trial on any technique applied thereafter renews the canary, thus forcing the attacker to start over again.

2.9 Conclusions

This paper considers the idea of renewing the canary at every new process creation stage (at fork time), and not only when a new image is loaded (at exec time). The current stack guard implementation generates a newly random canary for every new process image, i.e. when the `exec()` is called. All the child processes inherit the value of the canary from the parent. We propose renewing the canary value for every child. This is especially effective on multi-process networked servers where the main server forks processes to concurrently attend several clients.

We believe that an inaccurate statistical understanding of the re-randomisation effect may have discouraged other authors from considering the benefits of this technique. We show that re-randomising the canary improves protection against attacks several times over, particularly when combined with other commonly used protection techniques, at negligible cost.

The new technique is called 'RAF SSP' (renew after fork stack-smashing protector) and has the following properties:

- The RAF SSP strategy has a negligible overhead.

- The canary brute force attack, especially the byte-for-byte variant, is no longer possible.
- While the attack on the standard SSP follows a uniform distribution, the attack on the RAF SSP is a geometric distribution.
- The solution can be implemented by means of a preloaded share library, and so it does not require modifying the source code of the server or recompiling it. Furthermore, neither the operating system nor the system libraries require modifying.
- The RAF SSP has been validated with several network servers: apache2, lighttpd, proftpd and samba, without modifying the source code or recompiling.

Chapter 3

SSPMD: Stack-Smashing Protection for Mobile Devices

This chapter analyses how the Android architecture jeopardises the effectiveness of the SSP. The secrecy of the stack guard (canary) can be compromised easily, because the same value is used by all Android applications. The RAF SSP technique is adapted and validated for Zygote. A patch to the Android source code is provided.

Contents

3.1 Introduction

A decade ago, buffer overflows, and particularly the variant known as *stack-smashing*, was the most dangerous threat to computer system security. Over the last few years, several techniques have been developed to mitigate the possibility of exploiting this kind of programming fault [10, 11]. Stack-smashing protection (SSP), address space layout randomisation (ASLR) and No-eXecute (NX[1]) are widely used in most systems due to their low overheads, simplicity and effectiveness. When these techniques are correctly implemented they prevent or mitigate stack-smashing, execution of return-2-x or ROP programming and code injection, respectively.

Unfortunately, it is not always possible to implement these techniques correctly. Just to mention a few examples[2]: the NX requires hardware support, otherwise it cannot be efficiently implemented under some architectures, and the ASLR is partially implemented (not all memory areas are randomised or are randomised only at system boot) in many systems, including some versions [22] of Android. One of the main problems affecting SSP (even in systems where it is correctly implemented) is the *byte-for-byte* [20] attack.

A technique that is known to be effective, but which is nevertheless used incorrectly provides a dangerous false sense of security that can be easily exploited by attackers. The fault or weakness remains latent for a long period of time, which in turn allows the attacker to prepare multiple assaults and tools that effectively bypass barriers that are generally considered as unbreakable (or properly settled).

SSP covers a narrow range of faults (compared with the ASLR), i.e. only those faults related to stack-smashing, but it is very effective against them [23], by detecting and aborting the process before the attacker can redirect the control flow.

3.1.1 Implementation challenges

A great deal of research and investigation has been done regarding stack smashing prevention during the last decade, especially on servers and desktop platforms. Unfortunately, none of the major smartphone platforms uses it as initially designed. In essence, the same design and implementation of the SSP technique as the one used in desktop and server systems should carry over to mobile devices, but there is a key difference that makes the SSP weaker on smartphone platforms: they have been optimised to reduce

[1]Also known as data execution prevention (DEP) or Write or eXecute (W ^X).

[2]It is beyond the scope of this paper to present an exhaustive list of improper or partial implementations.

application launch time, power consumption and memory footprint. These optimisations have had a subtle impact on the execution of the applications and have invalidated one of the SSP's original assumptions: *the value of the canary is a secret random value which is different for each application.*

Android applications are clones (or forked processes) of a single launching process called *Zygote*. As a strategy to speed-up launch time and reduce memory footprint, all applications share a large amount of Zygote's process state: libraries, the Dalvik virtual machine and some initialised data, which unfortunately include the canary value. Therefore, all Android applications use the same random canary value, which is initialised at boot time and is not changed until the next reboot. This is an implementation weakness that can be exploited, in order to bypass the SSP. It is important to differentiate between *Android applications*, which use the Android framework (i.e. libraries and services) and are written in Java, and *native processes*, which are normal Linux processes, typically written in C/C++. The contributions of this paper are related to Android applications.

3.1.2 Our contributions

A summary of the main contributions of this paper is as follows:

1. We show how the current Android architecture breaks some core assumptions of the SSP, which in turn greatly reduces the effectiveness of this protection technique.

2. A new stack-smashing protector for mobile devices (SSPMD) technique is proposed, based on RAF-SSP [24] concepts, which addresses all the issues caused by the Android architecture.

3. The SSPMD prevents brute force attacks against the SSP and ASLR [25] on stack buffer overflows for Android applications when relaunched from Zygote.

4. Also, the SSPMD mitigates SSP vulnerability, presented by [26], due to the incorrect usage of the Android pseudo-random number generator.

5. The SSPMD has been implemented and evaluated on Android 4.2. The implementation shows that it can be implemented by adding five lines of code.

6. The evaluation shows that the SSPMD has a negligible overhead.

The rest of the paper is organised as follows. Section 3.2 provides specific background information on Android, and the threats and vulnerabilities caused by the Android framework in relation to the SSP technique are presented in section 3.3. Section 3.4 describes the proposed modification to the SSP, to overcome the deficiencies identified, while the implementation of the new technique (SSPMD) is presented in section 3.5. Section 3.6 evaluates the implementation, and the paper finishes with a discussion on the general applicability of the proposed technique and a number of conclusions.

3.2 Overview of Android

Android is an operating system, developed by Google, Inc., for mobile devices. While it borrows a good deal of platform code from the Linux operating system, its security model was built from the ground up, with the assumption that the device would be running a variety of untrusted or partially trusted applications. Isolation/protection between applications is achieved by executing each one in a separate process with a different UID per application. This can be seen as a different confinement memory area per application. A vulnerability in one application should not affect any others.

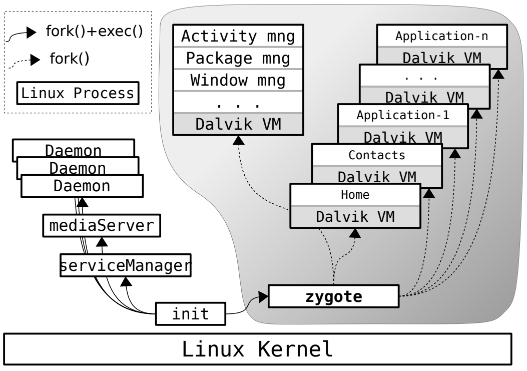


Figure 3.1: Android process tree.

Android is built on top of the Linux kernel with some specific device drivers and a “C” library for embedded devices, namely the *Bionic* library. Android applications are written in Java and executed in the Dalvik virtual machine (or ART, in recent versions, $\geq$4.4). The execution framework is composed of a set of servers, most of which are written in Java, which provide all kinds of high-level services to the applications.

Regarding the SSP implementation in the Bionic library, it uses a fully random canary (that is, all bytes of the canary are random values).

Zygote is an important Android process, used mainly to speed-up application launch. It is initiated at boot time with commonly used, shared libraries, application frameworks and the Dalvik virtual machine with some classes and resources that applications will need. Next, it waits for commands on a socket. When a new application is requested to be launched, Zygote forks itself creating a new process and loading the application code in the pre-warmed up virtual machine. Since most resources are already loaded in Zygote, the application can immediately begin executing. And thanks to the copy-on-write mechanism, most of the system resources are shared until they are modified. Therefore, all processes forked from Zygote (i.e. all Android Applications) use exactly the same copy of the system classes and libraries as well as **the same reference canary value**.

3.3 Threat Model

The SSP is effective as long as the value of the canary is unknown to the attacker. The variant of the SSP which zeroes one byte of the canary is also effective against overflows caused by incorrect string operations, even if the attacker knows the value of the canary, because the zero byte is interpreted as the string terminator, which stops any form of overflow at that byte. Nonetheless, the reduced range of values (2^{24} on 32-bits systems), in tandem with continuous advances in compiler analysis (i.e. GCC fortify features), has made this type of canary less effective[3].

We will focus on full-word random canaries: SSP is effective while its value is unknown to an attacker, and any form of information leak is a threat to the technique.

In this section, rather than a detailed explanation of how to bypass the SSP (the reader is referred to [20]), we will focus on the SSP weakness that affects Android systems.

As stated in the previous section, one of the key elements of the Android architecture is Zygote, which is the parent of all Android applications. This characteristic has the undesirable effect that all Android applications inherit the same reference canary value, which implies that any local Android application knows the canary value of any other application.

There are basically two types of canary leakage: brute force attack and direct observation.

[3]Note that it is risky to make this kind of statement, especially in the security field. Therefore, it must be treated with caution and considered only in the current context.

3.3.1 SSP brute force attacks

Different canary values are tried until the correct one is found. In order to build a brute force attack, four conditions must be met:

1. The attacker must guess the secret.
2. The attacker has to be able to decide whether it is a correct or an incorrect guess.
3. The guess can be repeated as many times as needed by the attacker.
4. The secret value must always be the same. That is, it must not change during the attack, otherwise tried and failed values cannot be discarded.

The first two conditions occur when there is a bug in the application and the bug manifests in such a way that the second condition can be applied.

The third condition is typically given on forking and pre-forking networking servers. In this case, the main server does not directly attend to client requests, but instead it forks child processes which are in charge of attending to clients. Each child inherits the socket from the client as well as most of the parent's state, which includes the reference canary value. The operation of Zygote has the same problem as forking servers, but it is augmented by the fact that not only the children of one single server share the same reference canary value, but all Android applications of the phone have the same value.

Also, considering that most users reboot a phone only when it is strictly necessary [27] (perhaps due to flight regulations, when installing major software releases, system hangs, run out of battery, etc.), the applications have the same reference canary value for very long periods of time, which increases exposure time.

At first glance, the fourth condition may seem to be a direct consequence of the third one, but it is interesting to note that if the secret value changes after a trial, then there is no way to build a brute force attack. An example of an application that does not meet the last condition is the SSH [28] suite, which improves its security by breaking this condition and using the sequence `fork() + exec()` when launching new clients. In x86_64 GNU/Linux most of the secrets (SSP and ASLR) are renewed after an `exec()`, but the drawback is the high temporal cost of the `exec()` operation.

Unfortunately, Android meets the fourth condition, even when the buggy application is not automatically re-forked or re-launched – as in is the case of a forking server – but id relaunched by the end-user manually.

Depending on the granularity of how the attacker can flood the buffer (word or byte overflow), there are two different brute force attacks that can be applied to Android applications:

Full brute force: the frame canary word is overwritten on each trial. If the guessed word is not correct, the application detects the error and aborts. The guessed value is discarded, and then the attacker proceeds with another value until all possible values have been guessed. The number of trials to bypass the SSP is $\frac{2^{32}}{2} = 2,147,483,648$ on average.

Byte-for-byte: this is a dangerous kind of brute force attack which consists of overwriting only one byte of the canary in each trial until the value of the target byte is found; the remaining bytes of the canary are obtained by following the same strategy. The system can be defeated with, at most, $\frac{4 \times 256}{2} = 512$ trials, which is a fairly low number.

3.3.2 SSP Direct disclosure

Every application running on an Android phone knows the system canary value, and so attackers can add a simple but useful ‘Trojanised’ application [29] (examples of such apps are lantern, notes takers or simple but appealing games) to ‘Google Play’, which sends to the attackers the value of its own canary value, jointly with other useful information,for example the ASLR memory map.Information can be sent directly from within the Trojanised application to the commander computer. But there are more subtle ways to do it, for example as a bug report which contains a stack dump along with other process information.

Note that attackers can introduce legal applications which do not cause any damage to the system or try to launch an attack on other applications but just obtain local secret information. These applications only access their own data and do not require highly suspicious phone permissions. Some people care about the permissions granted to an application, but in this case no Android permission other than internet communication is required to release the system’s canary value.

3.3.3 ASLR brute force attack

The implementation of the ASLR on Android suffers from a weakness similar to that of the SSP. Recently, a solution to this problem was proposed [30] which consists in randomising the offsets of the libraries at system update time. Even in the case of better ASLR implementations which randomise per boot instead of per system update, Android applications inherit their memory map from the Zygote process. This happens because the design of the Android system relies on this characteristic to reduce application launch time.

Unfortunately the offset where the libraries are loaded and the canary value are the same for all Android applications per boot, thereby allowing attackers to build remote brute force attacks to bypass first the SSP and later the ASLR.

3.3.4 Summarising

The execution environment in Android applications jeopardises the effectiveness of the standard SSP technique.

- Weak security control in the Google Play store makes it relatively easy to upload malicious applications, which in turn are installed by careless users. Therefore, contrary to desktop and server systems, local attacks on smart-phones (especially on Android) represent a main attack vector.
- Current Android SSP implementation is completely useless against local attacks, because the canary value is not a secret to local applications.
- Zygote, as well as other system applications (executed by the Dalvik VM), has the same broken SSP implementation.
- Remote attacks that have to bypass the SSP on a target application may first attack the weakest installed application, to obtain the canary value, and then use the obtained value against the real target application. This attack strategy makes exploitable some applications that otherwise would not be vulnerable.
- There is a very long exposition time. Once an attacker obtains the canary value, they can use it as long as the system is not rebooted, which may be a fairly long period of time.
- The obtained canary value can even be used against applications installed after the canary value has been leaked.
- Stopping, restarting or re-installing the application, does not change its canary value.

3.4 SSPMD

The SSPMD relies on the same SSP infrastructure as already implemented by Bionic and GCC, but the Zygote code is modified to **renew the reference canary on the child process right after the new process has been created (forked)**. It is important to note that the value of the reference canary of Zygote is left unchanged, and only the reference canary of the forked/cloned processes is modified.

In order to understand why this modification does not break the normal operation of the application, the following observations shall be considered:

- In most applications, after a `fork()` operation, the child process executes a flow of code which ends with an explicit call to the `exit()` system call, i.e. the child process does not return to the main flow of control but jumps to execute the specific child code, which in turn ends with a call to `exit()`.
- There is a single reference canary per process which is stored in a protected/separated area and initialised during process start-up.
- Integrity (i.e. checking the frame canary against the reference canary) is only done at the end of each function (or block of code), immediately before the returning instruction.
- Only the value of the frame canary of the current stack is checked against the reference canary.

The function where the canary is changed defines a ‘point of no return’. To be more precise, once the reference canary has been changed, any attempt to return from a function whose frame canary does not match the new one will abort the process. In other words, it is possible to return to previous functions once the reference canary has been changed, if – and only if – those functions do not have a frame canary. Therefore, the real ‘point of no return’ is defined by the first parent function, which effectively checks the integrity of the stack. Considering that only functions that declare local buffers are protected by the SSP, it may be possible to change the reference canary and return back several functions, if the necessary conditions are provided. This subtle detail in how exactly the SSP operates is important, because it greatly simplifies the implementation of the SSPMD technique on Zygote. Figure 3.2 shows graphically the division of the stack once the reference canary has been changed.

The application can call and return to any function freely, but it cannot return to the functions that called it originally.

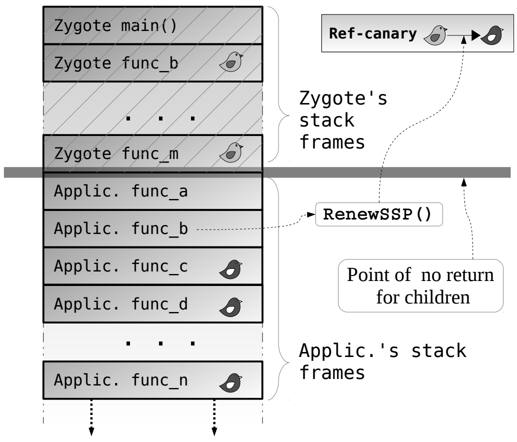


FIGURE 3.2: Application active stack.

Non-local jumping (i.e setjmp/longjmp) is another form of control flow which can disrupt the normal execution of a program. It is typically used as an exception mechanism to jump or restore back multiple levels of function calls, in order to continue from an initial safe state. Since `longjmp()` code does not check the stack integrity of the current and the destination functions, it can be safely used after a reference canary change. However, care must be taken if the destination frame of the destination function (or the previous stacked frames) contains frame canaries with the old value. The way to solve this problem involves storing the value of the current reference canary when the `setjmp()` is called, along with the rest of environment information, and then restore the reference canary to its original value when the `longjmp()` is later invoked. The value of the reference canary shall be considered a part of the execution context, and since it is not guaranteed to be constant throughout the execution of the process, it should be stored/restored when needed.

3.5 SSPMD on Android

This section analyses the specific needs of the Android platform when implementing the SSPMD, and it also presents the code for the proposed implementation.

In what follows we analyse the relationship between the application codes with respect to previous Zygote executions:

3.5.1 Application launch

There are three different phases involved in launching a new process: 1) process creation, 2) binding application and 3) launching activity. We will focus on the first one, because this is where the new application and the Zygote code depart from each other and where the canary should be renewed.

During this first phase, the `ActivityManagerService` receives a startActivity *intent* – if the process associated with the activity does not exist, it sends to Zygote a request to create it via a connection socket; upon reception of the request, Zygote forks a new process and instantiates the `ActivityThread` object, which starts the message loop by calling `Looper`. The next step involves attaching the process to the specific application and then finally launch the activity by calling the `onCreate()` function, which is the first call of the entire lifetime of the activity. We are only interested in the path of code executed from the `fork()` to the beginning of the application code.

The Zygote code that manages the creation of new processes is located in the file `libcore/dalvik/src/main/dalvik/system/Zygote.java`, which defines the `Zygote` class. This class provides, among others services, two public static methods to handle the creation of new applications and services: `Zygote.forkAndSpecialize()` and `Zygote.forkSystemServer()`. The sequence of calls from high-level functions to the instruction which performs the system call (jumping to kernel code) is summarised in Table 3.1.

Lang	Function name
Java	`Zygote.forkSystemServer()/forkAndSpecialize()`
Java	↳ `nativeForkSystemServer()/nativeForkAndSpecialize()`
C++	↳ `...Zygote_forkSystemServer()/Zygote...Specialize()`
C++	↳ `forkAndSpecializeCommon()`
C++	↳ `fork()`
Asm	↳ `__fork()`

TABLE 3.1: Call sequences.

These code sequences are executed once on the call sequence (by the parent process) and twice when returning (both the parent and the child process).

A detailed analysis of the code of these two sequences of calls shows that none of these functions is SSP protected, because they do not declare any local buffer. Therefore, it is safe to change the reference canary at any point during the execution of this sequence. In the case that any them are protected by a canary, due to a future change in Android compilation options (e.g. compiler flags are changed to the more secure `stack-protector-all` or `stack-protector-strong`[4] flag), then they shall to be compiled with the `no-stack-protector` flag. A simple modification of the build scripts would fix this issue.

Once started, the application acts as a server that executes callbacks, and so parent functions are never returned from them. This behaviour can be viewed as if there are two separate stacks – as shown in Figure 3.2. The upper part of the stack contains stack frames with the old canaries, and the bottom part is the live stack of the application, which uses the new canary.

3.5.2 Application termination

During the normal execution of an Android application, the functions of the old stack never retur. We analysed how the applications terminate, to find out whether or not the application returns to the old functions.

Processes are activity containers, and their creation or destruction is controlled by the kernel. Android's execution model does not consider the termination of an application by calling `exit()` explicitly, and its full life-cycle is beyond the scope of this work, but for our purposes it is enough to know that processes can terminated in either of the following two modes:

a) An application can call the method `Process.killProcess()` if the process is part of the application, or it can be killed by others if it has the '`android.permission.KILL_BACKGROUND_PROCESSES`' permission.

b) Some versions of Android used a queue that keeps track of which applications have not been used. If the OS starts to run out of memory, it will kill an application (according to some metrics).

In both cases, the process ends by means of a signal. It does not return to any saved stack/environment, which meets SSPMD requirements.

3.5.3 Exception handling

Although Android is compiled using the C++ compiler, the code is mainly 'C'-compatible. Fortunately, the exception handling of C++ (i.e. try-catch blocks) is not supported by the Bionic library, which forces one to check

[4]`stack-protector-strong` is still a feature not available in the stable version of the GCC compiler, as of writing this paper.

errors and exceptions, by using explicit conditional constructions. Therefore, the stack is never unwound, due to a raised exception. This restriction causes the native code of Zygote (which is affected by the SSPMD) to be very procedural and sequential. Also, there are no calls for the setjmp/-longjmp functions. On the other hand, the Java side of Zygote uses all the bells and whistles of the language, in which case exception handling is compatible with the SSPMD.

3.5.4 Modifications to Zygote

Once the impact of SSPMD on Android has been analysed, the implementation is straightforward. We implemented the SSPMD for Android 4.2 (repository branch: `android-4.2_r1`) Jelly. We chose this version of Android because it supports both smartphones and tablets.

The first step involves defining or giving access to a function which changes (re-randomises) the reference canary. We can reuse the already implemented function `__guard_setup()`, which initialises the reference canary with a random number from `/dev/urandom`. Rather than exporting this function we preferred to export a dedicated function (called `renew_ssp()`), for clarity.

The code added to the Dalvik machine required only one call to renew the reference canary (`renew_ssp()`) in the function `forkAndSpecialize-Common()` located in the file `dalvik_system_Zygote.cpp`, as shown in listing 3.1.

The function `forkAndSpecializeCommon()` is called from the functions `Dalvik_dalvik_system_Zygote_forkSystemServer()` and `Dalvik_dalvik-_system_Zygote_forkAndSpecialize()`. These functions are used to launch new children for system services and general-purpose applications, respectively.

3.5.5 Implementation discussion

The modifications introduced by the SSPMD are not architecture-dependent, so there are no restrictions on using our proposal on other hardware supported by Android, such as MIPS or x86.

A key implementation issue is the source of random numbers. The function `__guard_setup()` reads four bytes from the `/dev/urandom` device. Urandom then produces an unlimited stream of random bytes, using a pseudo-random number generator, based on the internal entropy pool of Linux. Although SSPMD consumes more random numbers than the SSP, it is only

```
static pid_t forkAndSpecializeCommon(...)
{
    ...
    dvmDumpLoaderStats("zygote");
    pid = fork();

    if (pid == 0) {
        int err;
        /* The child process */

#ifdef HAVE_ANDROID_OS
+++     renew_ssp();
        extern int gMallocLeakZygoteChild;
        gMallocLeakZygoteChild = 1;
        ...
   return pid;
}
```

LISTING 3.1: `dalvik/vm/native/dalvik_system_Zygote.cpp`

four bytes per application, which can not be considered a dangerous drain of entropy. Also, we need to note that this is the default consumption rate on conventional systems, where applications are launched using the `fork()+exec()` pair.

This implementation exports the `renew_ssp()` symbol. This way, any native application compiled against the new library will be able to call that function at will. This function does not receive any parameter from the user, so it is impossible to reset (or set to a known value) the value of the reference canary. Therefore, security is not vulnerable when renewing the value of the canary at any moment during the execution of the process, as long as the already stacked stack canaries are not checked. A more general discussion about how and why to change the reference canary can be found in [24].

It is important to note that the old reference canary was used intensively by Zygote before forking. That value was pushed and popped from the stack multiple times while calling and returning from functions. Therefore, the stack of the child may still contain a copy of that value. These garbage values may reside in any location on the stack (downward or upward) and may be observed by a malicious application. Since Zygote does not change its own reference canary, the malicious application may be able to read the current canary of Zygote, albeit not the reference canary of the rest of the applications. The solution to this issue is to also change the value of the reference canary on the Zygote after a new process has been created. An analysis of how and where the canary of the Zygote shall be changed is beyond the scope of this paper, but we shall mention here that it can be done by following a similar approach to the one used to protect the applications.

Another interesting aspect to consider regarding the simplicity of the implementation is that it does not change the logic of Zygote, which greatly simplifies the maintainability on future versions.

3.6 Evaluation

The following aspects were evaluated: 1) the correctness of the modification, 2) overheads, both spatial and temporal, 3) portability and 4) effectiveness.

The correctness of the implementation was evaluated by running the system and reading the values of the canaries for the Android applications in both the original system and the one modified with the SSPMD. An overhead is only created as a result of the cost of reading four random bytes during application launch, and there are zero overheads during the execution of the application.

The evaluation of SSPMD, i.e. its effectiveness, was analysed analytically, by comparing the operation of the current implementation with the new SSPMD. A detailed evaluation of the stack guard technique is beyond the scope of this paper.

3.6.1 Verification of the implementation

The implementation was tested by reading the values of the reference canaries on the original version of Android and then on the modified version.

The value of the reference canary can be read directly from the memory of the process, through the `/proc/<pid>/mem`. Bionic, the reference canary, is a global variable named `__stack_chk_guard`. In our example, it is located at the offset `0x4b228`.

The results relating to executing the inspector program are listed in Table 3.2. As expected, all of Zygote's children have the same reference value on a standard system but different values when using the SSPMD modification.

Table 3.2 also shows the canaries of native processes (those not launched by Zygote). In this case, the canaries are different because they are processes with a new binary image loaded by an `exec()` syscall.

Android application	**SSP**	**SSPMD**
`zygote`	`0x7852ee0c`	`0x20cf270d`
`system_server`	`0x7852ee0c`	`0xb2368f9d`
`com.android.phone`	`0x7852ee0c`	`0x96c40065`
`com.android.music`	`0x7852ee0c`	`0xc0a6c73c`
`com.android.mms`	`0x7852ee0c`	`0x34ff9aa8`
`com.android.launcher`	`0x7852ee0c`	`0x94ff1193`
`com.android.contacts`	`0x7852ee0c`	`0x61c22d06`
`com.android.calendar`	`0x7852ee0c`	`0x77cdb8a5`
`android.process.media`	`0x7852ee0c`	`0xe5f7ef65`
`/system/bin/sh`	`0xff0422c9`	`0xe95b0903`
`/system/bin/rild`	`0x7a0f3f72`	`0xb6b3e8d1`
`/system/bin/mediaserver`	`0xab5aa3f7`	`0x7c8879eb`
`/system/bin/keystore`	`0xae444921`	`0xc05866d3`
...	...	...

TABLE 3.2: Reference-canaries with both techniques.

3.6.2 Memory footprint

The implementation of the SSPMD relies on the already existing infrastructure of the SSP and needs neither global nor local stack frame additional storage. Our implementation applies and exports the `renew_ssp()` function, which is just a proxy to `__guard_setup()`, from *bionic* and adds a single call to this function in Zygote code. The modified Zygote function is not in the executable itself but in the shared library `libdvm.so`.

The amount of code added is so small that the default optimisation of function alignment to a 32-byte boundary may hide the size of this additional code.

The size of the Dalvik virtual machine program is not increased at all in the ARM processor, due to alignment padding, which means that the SSPMD technique can be used on a mobile phone with zero memory overheads. And in the case of the x86, the global cost of the SSPMD is a total of 25 bytes. Note that this value is independent of the number of applications executed in the phone.

3.6.3 Temporal overhead

The temporal overhead is caused by the call to renew the canary on the child process after the fork operation, which is called only once per application. The rest of the execution of the application has zero overheads.

It is a continuously evolving sector in which devices are surpassed in a matter of months. It is therefore pointless to try to find a representative device for running performance tests, so we selected two phones according to their availability, namely the Samsung Galaxy S4 mini and the Huawei U8650 Sonic. Table 3.3 summarises the average cost of calling 10.000 times the function `renew_ssp()` on several devices.

Model	**Mean time**
Tablet, Asus Nexus7-1B	13
Huawei U8650 Sonic	26
Samsung Galaxy s3-i9300	11
Samsung Galaxy S4 mini	38

TABLE 3.3: Cost of renewing the canary (μsec).

Although the S4-mini is faster than the Huawei U8650, it took 38 μs versus the 26 μs of the Huawei U8650, because the kernel on that platform implements the SELinux facility, which adds an extra overhead to each system call, including the three calls (`open()` , `read()`, `close()`) needed to read from `/dev/urandom`.

3.6.4 Portability

Although the implementation of the SSP is highly processor- and compiler-dependent, the SSPMD is not. Fortunately, neither the compiler nor the supporting library functions have to be modified, as all code modifications have been done in 'C' and in the generic part of the libraries. No platform-specific code has been added. Therefore, SSPMD is fully available to current platforms (ARM, MIPS and x86) and will be automatically available on new portings. Obviously, this transparency in the implementation greatly simplifies the maintainability on new releases for the same platform. SSPMD does not break any assumption or impose complex requirements or limitations on the Android architecture.

The only limitation is that once the reference canary has been changed, it is not allowed to return from previous stacked functions, if those functions check the canary. It should be considered that it is extremely rare to return back to the parent code after a `fork()` operation, and as far as the authors know, child processes execute another flow of code which always ends with a call to `exit()`. In order to validate this affirmation we conducted an experiment which involved modifying the implementation of the Glibc `fork()`, to always renew (on the child) the canary. A complete Ubuntu 13.10 distribution, using this library, was used seamlessly.

Therefore, we can consider that the restriction required by SSPMD – that child processes must not return to parent functions – is not a limiting or an unacceptable requirement, because it is normal default behaviour fr all analysed applications.

3.6.5 Vulnerability coverage

All the issues described in section 3.3 are settled by the SSPMD. The following is a brief discussion on each one and how the SSPMD addresses them:

SSP brute force: it is impossible to implement a SSP brute force attack against Android applications (non-forking servers) to bypass the canary, since the canary is renewed on a per application basis when the SSPMD is used. An application crash implies that a new process for the same application has been launched, and consequently it has a different canary. The only choice for the attacker is to try a probabilistic attack.

SSP byte-for-byte: it is interesting to note that the very dangerous SSP byte-for-byte attack[5] (because of the reduced number of trials) cannot be used against the SSPMD on Android applications. In this case, the attacker is faced with a full-word probabilistic attack, which is quite discouraging.

ASLR brute force: it is impossible to implement a stack buffer overflow ASLR brute force attack against Android applications. When the SSPMD is used, the SSP brute force attacks cannot be employed, because any incorrect guess will re-launch the application and the canary will be renewed. In this scenario the attacker is not able to perform a brute force attack against the ASLR but a probabilistic attack against ("trial-and-test") against both secrets at once, which has also a multiplicative effect on the number of trials. Therefore, the SSPMD protects against remote ASLR brute force attacks on stack buffer overflows.

Direct disclosure: if an application has a vulnerability which allows one to directly read the value of the canary, the obtained value can only be used on the same application – and only until the application has been restarted. Unlike current SSP, with SSPMD a crash in an application renders useless the guessed value.

Local attacks: it is no longer possible to know the value of the canary from another application running on the same system. Contrary to the original SSP, which can be considered as a defeat technique when considering local attacks, the SSPMD has the same level of effectiveness in relation to local attacks as remote ones.

[5] Although byte-for-byte is a form of brute force attack, we have considered it separately, due to its singularity.

Weak randomness: the SSPMD mitigates the vulnerability caused by generating the Zygote frame canary, before the PRNG of the Android kernel is properly set-up [26]. By the time the applications are launched, the quality of the PRNG has already been corrected, and so the SSPMD is an effective workaround for this issue.

Table 3.4 summarises in a few words the achievements of the SSPMD with respect to the current implementation.

Threat/Issue	**SSP**	**SSPMD**
SSP full brute force attack:	Yes	No
SSP byte-for-byte attack:	Yes	No
ASLR brute force (on stack):	Yes	No
Direct disclosure bypasses:	All apps.	The affected app.
Local attacks are:	Trivial	Same as remote
Canary exposed until next:	Reboot	App. relaunch

TABLE 3.4: Android SSP vs. SSPMD summary.

3.7 Discussion

It can be argued that Android applications are not native applications but are instead byte code interpreted by Dalvik. Unfortunately, Android applications use native code through JNI. Note that the SSP technique is only applicable to native code; in fact, many libraries are written in C/C++ and export their services via JNI to Java applications. Some application parts are written in C/C++, to overcome Java limitations, for example in order to access system services that are not available otherwise (for instance, to interact with POSIX pseudo-terminals), to speed-up critical parts or to reuse existing C/C++ code. SSP protection takes place in all of this native code, used by Android applications.

Another aspect to consider is the applicability of the SSPMD. Although it may seem that the code in which the SSPMD is used must meet very specific and somewhat odd conditions, a deep analysis of the code involved in how the `fork()` syscall is typically used reveals that most real applications meet those conditions by default. That is, the SSPMD can be used with minor modifications to Zygote, because it has been coded following standard programming patterns.

In [24], the authors carried out an experiment to test the impact of the more general RAF-SSP technique on a full desktop system. The `fork()`

function of the Glibc library was replaced by a custom `fork()` which always renews the reference canary on all children processes. A complete Linux distribution, using the modified library, ran smoothly. We cannot conclude that the RAF-SSP technique could be applied as a simple drop-in replacement for the fork, but it is very likely that a simple inspection of the code – and in some cases a small modification – would be sufficient for its use.

The SSP technique is under active development. In the latest version of Android, KitKat 4.4.4_r1, the initialisation of the canary has been moved into the bionic core constructors. Also, regarding the GNU GCC suite, Google recently implemented the `stack-protector-strong`, which represents a balance between performance and coverage.

The SSPMD technique can be considered as another defensive measure which can be included in software, in a similar way to other measures such as drop privileges, assertions, data canonisation, etc.

3.8 Conclusions

Stack buffer overflow is one of the most dangerous vulnerabilities in computing, because when successfully exploited, it gives direct access to the control flow of the program. Thanks to non-executable data, aligned with the SSP and the ASLR mitigation techniques, the exploitable nature of this type of vulnerability has been greatly reduced.

The SSP is a technique which covers a narrow range of vulnerabilities, namely only those caused by stack buffer overflows that occur on vectors located in the stack; however, it is very effective when correctly implemented. The SSP technique relies on keeping secret the value of the stack guard (canary), which is a random value that must be unknown from outside of the application.

The Android architecture violates a key design principle of the SSP, in that the canary must be a secret per individual application. The SSPMD restores back the effectiveness of the original SSP by setting a different canary for each Android application.

The SSPMD is not intrusive, in the sense that it can be implemented just by adding one line of code in Zygote, and it is binary-compatible with all current and future Android applications. That is, applications do not need to be upgraded to benefit from this improved protection. Finally, the new technique has been validated on a real platform.

Chapter 4

Method for Preventing Information Leaks in the stack-smashing protector technique

A generalisation of the renewSSP technique, described by following the rigorous and precise format of a patent request, is presented in this chapter.

The patent extends the scope of the renewSSP to define the conditions in which the reference canary can be renewed transparently, and when and how the canary can be renewed but must later be restored to continue the normal execution of the process.

Contents

4.1 Abstract of the Disclosure

A method for hardening the stack-smashing protector (SSP) technique, which prevents information leaking from the protecting guard, is disclosed herein. The reference stack guard secret value is renewed at one or more selected time points during the execution of the application. The technique is non-intrusive and has a negligible run-time cost (both spatially and temporally). The technique reuses the SSP infrastructure, and it does not need to recompile code or modify the binary image of the application. The method prevents any kind of brute force attacks against the SSP technique and most info leaks affecting the canary guard.

4.2 Background

A decade ago, buffer overflows, especially stack smashing, were the most dangerous threats to computer system security. Over the last few years, several techniques have been developed to mitigate the ability to exploit this kind of programming fault. Stack-smashing protector (SSP), address space layout randomisation (ASLR) and Non-eXecutable (NX) are widely used in most systems, due to their low overheads, simplicity and effectiveness.

Following the classic measure/counter-measure sequence, a few years after the introduction of each protection technique, a method to bypass or reduce its effectiveness was published. SSP can be bypassed using brute force attacks or by overwriting non-shielded memory, the ASLR can be bypassed using brute force and the NX, which effectively blocks the execution of injected code, can be bypassed using ROP (return-oriented programming). In spite of existing counter-measures, these techniques are still effective protection methods, and in some cases they are the only barrier against attacks until software is upgraded to remove the vulnerability.

Unfortunately, the forked and pre-forked networking server architectures are especially prone to brute force attacks, as all of the children processes inherit/share the same memory layout and the same canary as the parent process. Consequently, an attacker can try – in bounded time – all possible values of the canary (for SSP) and memory layouts (for ASLR) until the correct ones are found. There is a very dangerous form of SSP attack, called byte-for-byte, whereby the attacker tries each byte of the canary independently, which permits him to determine the value of the canary with just a few hundred trials, and as a result a system can be defeated in just a few seconds.

Another area where the standard SSP technique is not as effective as originally designed is the software architecture in which a single (launcher) process prepares the execution environment of the children applications by pre-linking, pre-loading and setting up the runtime environment. This architecture is typically used to speed up launch time and to reduce resource usage. The canary value of the SSP (the secret) is inherited by all the children processes, and so all of the children share the same secret. An information leak, accidental or intentional, from any of the children may therefore compromise the security its siblings or even the base system.

For the sake of clarity, we will assume that the stack grows downwards, i.e. from higher to lower addresses, but the invention also applies to systems which implement upward-growing stacks or any other way of protecting sensitive data on the stack from vector (or other data) overflow, for example, but not limited to, the stack protection technique disclosed in patent application Ser. No. US 13/772,858.

The value of the canary is a random value computed during process initialisation. In order to bypass SSP protection, the attacker must know the current canary value. Whilst the value remains kept secret, the attack will be prevented by the SSP. In some implementations, the canary value is a word with all bytes random except one, which is zeroed. The zero byte is used to prevent the possibility of exploiting an error caused by incorrect string handling. Our invention is not limited by the way in which the random value is computed.

Since the value of the canary is not a constant but a random value chosen when the program starts, this value, called the 'reference-canary', has to be stored somewhere in the program memory or in a dedicated processor register, if available. For example, in x86-32 and x86-64 architectures the reference canary is stored in a special data segment which is not accessible as a normal variable and cannot be easily overwritten or read.

4.2.1 Known patent documents

US 6941473 B2, entitled 'Memory device, stack protection system, computer system, compiler, stack protection method, storage medium and program transmission apparatus', discloses a method that uses a guard value, or canary, to protect both the return address and the previous-frame-pointer from the local function buffers. This patent is an extension or adaptation of the work presented in 'Automatic Detection And Prevention Of Buffer Overflow Attacks', by Crispin CoWan, Calton Pu, David Majer, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Quan Zhang, the 7th USENIX Security Symposium, San Antonio, Tex., January 1998, to protect the previous-frame-pointer. This innovation was later refined by the ProPolice strategy, which arranges the content of the stack to avoid local scalar variables being overwritten by local buffers. None of these improvements addresses the problem of canary value leaks, which is the novelty of the present disclosure. US 6941473 B2 is hereby incorporated by reference into the specification of the present invention.

US 6578094 B1, entitled ‘Method for preventing buffer overflow attacks’, discloses a method involving a called procedure determining an upper bound that may be written to a stack-allocated array/buffer, without overwriting the stack-defined data. Before data are written to the stack, the upper bound is checked, which thereby prevents overwriting said data. The present method does not check for an upper bound before writing data to a stack. US 6578094 B1 is hereby incorporated by reference into the specification of the present invention.

US 7581089 B1, entitled ‘Method of protecting a computer stack’, discloses a method of having two stacks: a normal stack and a second one to which the return addresses are copied. Both stacks are automatically compared and re-synchronised at each return. The present method does not use a secondary stack, and the return address is not checked or validated. US 7581089 B1 is hereby incorporated by reference into the specification of the present invention.

US 7660985 B2, entitled ‘Program security through stack segregation’, discloses a method of having two stacks: a normal stack, which grows downward, and an inverse stack, which grows upward. Items on the stack data structure are segregated into protected (frame pointers and return addresses) and unprotected (function parameters and local variables) classes. The present method uses a single stack and also does not modify the layout of the stack. US 7660985 B2 is hereby incorporated by reference into the specification of the present invention.

US 7086088 B2, entitled ‘Preventing stack buffer overflow attacks’, discloses a method and system for preventing stack buffer overflow attacks by encrypting return addresses prior to pushing them onto the runtime stack. When an encrypted return address is popped off the runtime stack, the computer system decrypts the encrypted return address, in order to determine the actual return address. The present invention does not alter the return address. US 7086088 B2 is hereby incorporated by reference into the specification of the present invention.

US 8631248 B2, entitled ‘Pointguard: method and system for protecting programs against pointer corruption attacks’, discloses a method for protecting against pointer corruption by encrypting a pointer. The encrypted pointer is decrypted before the pointer is used. The present invention is not directed at encrypting pointers. US 8631248 B2 is hereby incorporated by reference into the specification of the present invention.

Both US 7467272 B2, entitled ‘Write protection of subroutine return addresses’, and US 8028341 B2, entitled ‘Providing extended memory protection’, disclose two methods of moving return addresses to the processor and providing a method of write-protecting return addresses to make them non-accessible. Both methods require the modification of the processor, or the memory management unit (MMU), so that the execution platform has the ability to lock (write-protect) very small blocks of memory. The present method can be used with existing hardware. Both US 7467272 B2 and US 8028341 B2 are hereby incorporated by reference into the specification of the present invention.

CN 1294468 C, entitled “Dynamic stacking memory management method for preventing buffering area from overflow attacking” discloses a method for preventing stack buffer overflows by dynamically adding a random number of padding bytes between the stack buffers and the return address. So that an attacker can not accurately determine the location of the return address. The present invention does not modify the layout of the stack, and so it can be used transparently on current systems. CN 1294468 C is hereby incorporated by reference into the specification of the present invention.

4.2.2 Known patent application documents

Patent application US 2013/0219373 A1, entitled “Stack overflow protection device, method, and related compiler and computing device”, discloses a method of splitting the code of at least one function into code which contains string manipulation (which is supposed to be prone to buffer overflows) and code without that behavior. The stack protector guard is used only in the region with the string operation, which is a clever way to reduce the overhead on the stack protector technique, but it does not prevent against guard leaks. Patent application US 2013/0219373 A1 is hereby incorporated by reference into the specification of the present invention.

Patent application US 2004/0168078 A1, entitled ‘Apparatus, system and method for protecting function return address’, discloses a method of protecting against stack overflow by storing the return address and the stack pointer in a separate stack. The return address is evaluated before executing the return, to check if it is a valid return address. No read or write function is permitted on the separate stack, thereby making this second stack secure. The method of US 2004/0168078 A1 guards against stack overwrite, which requires extra memory space to back-up sensitive information (return address and stack pointer). Patent application US 2004/0168078 A1 is hereby incorporated by reference into the specification of the present invention.

4.3 Summary of the Invention

The goal of the present invention is to overcome the deficiencies found in stack-smashing protection (SSP) techniques.

When existing SSP techniques are employed in applications where multiple processes share (inherit) the same canary value, the secrecy of the canary may be treated by the dissemination of the secret canary among multiple processes. A fault or information leak in any of the process that share the same canary value may compromise the security of the whole system.

The present invention is applicable, but not limited to, networking applications in which several processes are used to attend to client requests (forking and pre-forking architectures), and also execution platforms where client applications are launched from a parent process which pre-loads libraries and prepares the execution environment of the children. One of ordinary skill in the art will know other types of execution frameworks in which the same canary value is also shared among different execution entities.

The present invention is also applicable to a single process that guards against potential canary value leaks, by making the leaked canary value useless.

The present invention is effective against canary leaks. For example, but not limited to: all forms of brute force attacks against the canary, direct information leaks, format string vulnerabilities, improper memory dumps or malicious code that intentionally reveal the canary value.

The present invention identifies, in some embodiments, functions that are relevant to protecting the secrecy of the canary value and how to effectively change the value of the canary for said functions such that potentially stolen information is useless to an attacker.

Although the present invention is directly related to the SSP technique, when the disclosed invention is used in combination with other commonly used protection techniques, such as ASLR and NX, it greatly increases by several orders of magnitude the difficulty involved in building a successful brute force attack.

The present invention is an advancement in the art of protecting applications from stack buffer overflow attacks by applying the following:

- It relies on the existing SSP infrastructure.
- It provides an effective protection mechanism against canary value leaks. In particular, it is no longer possible to perform any kind of brute force attack.

- It preserves backward compatibility, because the layout and content of the stack are not modified.
- It can be used on most applications and does not require one to either modify or recompile the application code.
- It does not use extra memory space in most cases, or only a few computer words in rare cases.
- It provides a solution that does not disrupt debuggers used in software development cycles.
- The overhead introduced by the present invention is negligible.

4.4 Detailed Description of the Invention

In order to understand the operation of the disclosed invention, the following general observations from the previous state of the art should be considered:

- Most applications, especially networking servers, after a fork operation, ensure that the child process executes a different flow of code which ends with an explicit call to the exit system call. That is, the child process does not return from the function that started the child code.
- Each child process of a network server defines an error confinement region. That is, any error that occurs in a child process does not affect the correct operation of the parent or other sibling processes, as long as the temporal and spatial isolation is honoured.
- Although there are several variants of the SSP technique, most implementations use a single reference canary **100** per process, which is saved in a protected area and initialised during the process start up.
- The reference canary **100** is copied in the stack frame **101** between the return address and the buffers, known as the 'frame canary' **103**. Depending on some compilation optimisations, not all stack frames are protected with a frame canary.
- Some SSP variants may implement slightly different versions of the basic mechanism, which does not invalidate the applicability of the disclosed invention.

- Stack integrity (comparing the reference canary **100** with the frame canary **103**) is only carried out at the end of each function, or block of code, immediately before the returning instruction or on leaving the block of code.
- Only the value of the frame canary of the current stack frame is compared with the reference canary.

The present invention consists of renewing the value of the reference canary of the process for selected functions, or blocks of code, during the execution of the process. There is only one single reference canary for each process. Our invention does not use a secondary stack to hold copies of the frame canaries, and it relies on the same infrastructure as applied in the SSP method.

Typically, there is at most one vulnerable function or vulnerable block of code per process. It is quite odd to have multiple buffer overflow exploitation functions in the same process. Therefore, from the point of view of the attacker, there is only one frame canary to defeat, which is that of the vulnerable function. Consequently, there is little benefit in randomising the canary in any function apart from the vulnerable one.

There are some special functions where the reference canary can be renewed and not restored, following which the program can continue its execution normally. An example of this type of function, but not limited to it, is the code executed by child processes right after their creation (for example, fork and clone), which matches the concept of error confinement region. For example, each client request is attended by the child process of a networking server when it is configured as a forking server.

Although it is possible to check manually that a function never returns, most compilers provide the `noreturn` function attribute, which declares a function as non-returning. The compiler generates more efficient code and checks (at compile time) whether or not the function honours the desired behaviour.

More generally, it is possible to renew the value of the reference canary at any time during the execution of a program, as long as the reference canary is restored to its previous value before the stack frames holding old canary values are checked.

In what follows, the terms 'function' and 'stack-frame' are used interchangeably. The former is an active element, and the latter is the passive data structure which supports function execution. Depending on the context in which the term is used, it is more natural to use one or the other, but in both cases it refers to the same concept.

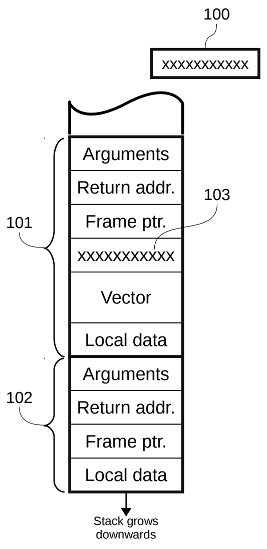


FIG. 1

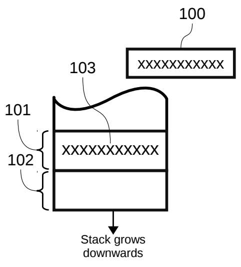


FIG. 2

FIG. 1 and FIG. 2 outline the content of a typical stack in two different forms: a detailed stack (FIG. 1) and a simplified stack (FIG. 2). FIG. 1 shows a stack with two frames **101, 102** filled with some example content on each one: Arguments, Return addr., Frame Ptr., etc. The frame **101** has a frame canary value **103**, and the frame **102** does not contain a frame canary. The present invention only depends on the values of the reference canary **100** and the frame canary **103**. FIG. 2 shows the same stack as FIG. 1 with two frames **101, 102**, though only the frame canary **103** is displayed. Those who are familiar with the state of the art will appreciate that our invention can be used with other stack frame contents and layouts.

In the rest of this document, we will use the simplified stack representation.

Our invention does not impose any restrictions on which stack frames shall be protected by a canary or which ones shall not be protected. Our invention is independent of the exact implementation of the SSP, and it can be used with any variant of the SSP.

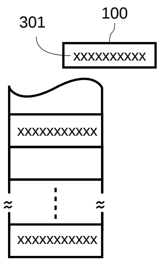


FIG. 3

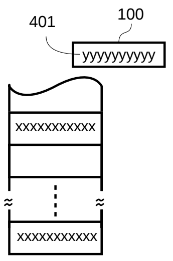


FIG. 4

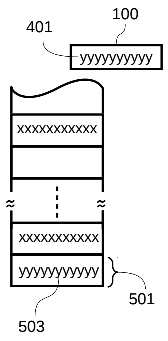


FIG. 5

Referring to FIG. 3, FIG. 4 and FIG. 5, they represent the state of the stack at three different points in time upon renewing the reference canary. FIG. 3 represents the stack and the reference canary **100** and its value **301** before the reference canary is renewed. FIG. 4 represents the stack and the reference canary **100** right after its value has been renewed **401** (from the value **301** to the new one **401**). FIG. 5 represents the state of the stack after a new frame **501** has been created using the renewed reference canary. The frame canary value **503** of the new frame **501** uses the renewed reference canary value **401**.

FIG. 6 to FIG. 11 represent the content of the stack for six different types of function. The stack frames of the functions where the reference canary **100** is renewed are marked as dashed boxes **601, 701, 801, 901, 1001** and **1101**. All figures represent the state of the stack after the reference canary **100** has been renewed. Therefore, the frame canary of the functions previous to the dashed as well as the dashed one contain the old canary

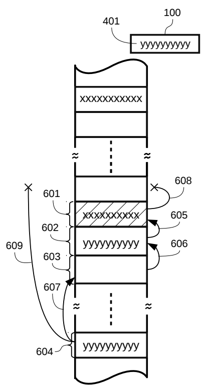


FIG. 6

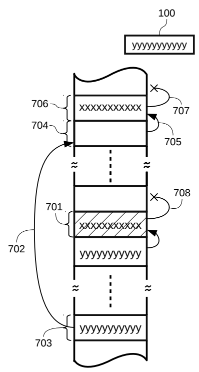


FIG. 7

value **301**, if any, and posterior frame canaries have the renewed value **401**, if any.

FIG. 6 represents the content of the stack when a non-returning function, the dashed stack frame **601**, has called some nested functions **602** to **604**. Each nested stack frame may (**602, 604**) or may not (**603**) have a frame canary. During the execution of the non-returning function **601**, called a 'type 1' function, the reference canary **100** can be renewed. The

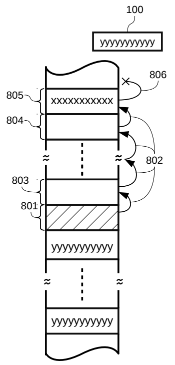


FIG. 8

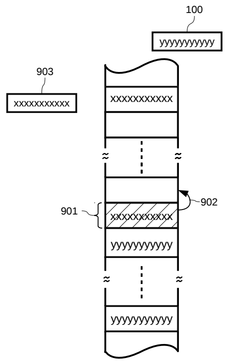


FIG. 9

frame canary value of the following functions (said nested ones **602, 604**) will be the new reference canary value **401**. These nested functions can make normal returns, with **605** or without a **606** SSP canary value check. They are also allowed to make non-local jumps **607** to functions within the nested region. A type 1 function must not return to the parent caller **608**. Furthermore, neither type 1 nor its nested functions **602** to **604** can make non-local jumps **609** to any parent function of this type 1 function.

FIG. 7 represents how our finding is used on 'type 2' functions. A type 2 function **701** does not return **708** but is allowed to make non-local jumps **702** from itself **701** or from a nested function **703** to a parent function of said type 2 function. The destination function of the non-local jump **702** can be any function **704**, for which none of its return-reachable **705** functions checks the frame canary value. Therefore, the functions that check the old frame canary **701**, **706** must not return **708**, **707**.

FIG. 8, shows how our finding is used on 'type 3' functions. A type 3 function **801** may return **802** but does not check the frame canary integrity neither itself **801** nor on any return-reachable parent functions **803, 804**. Parent functions which check the frame canary **805** cannot return **806**.

FIG. 9 represents how our finding is used on 'type 4' functions. A type 4 function **901** renews the reference canary **100** and eventually returns **902** by checking the frame canary. A type 4 function should save the original value of the reference canary **301** in a designated location **903** and restore it back (copy the old value from the saved reference canary **903** in the reference canary **100**) before the current frame canary is checked **902**. The original reference canary value must be saved by said type 4 function or any of its parent functions.

The saved reference canary is represented as a global variable **903** for clarity, but it is not limited to it. The saved reference canary may be saved as a local variable, on a dedicated memory segment, on a processor register or at any other retrievable location.

FIG. 10 shows how our finding is used on'type 5' functions. A type 5 function **1001** renews the reference canary and eventually returns **1002**. The stack check is not done by type 5 function, but there is at least one parent function **1003** which checks the integrity of its stack **1004**. The parent function **1003** which checks stack integrity **1004** should restore the reference canary to the original value, by copying the old value from the saved reference canary **903** in the reference canary **100**. In order to be able to restore the reference canary, it has had to be saved, in a designated location **903**, before it is renewed. The original reference canary value must be restored by said type 5 function or any of its parent functions up to the function **1003** which checks the frame canary.

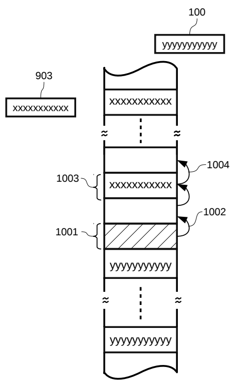


FIG. 10

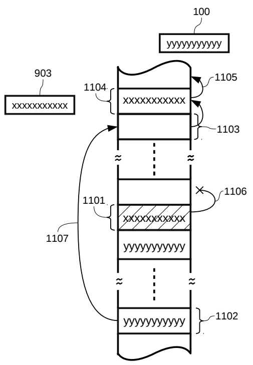


FIG. 11

FIG. 11 represents how our finding is used on ‘type 6’ functions. A type 6 function **1101** does not return to its caller **1106** but makes a non-local jump **1107** from itself **1101** or from a nested function **1102** to a parent function **1103**, for which said parent function **1103** or a previous caller function **1104** checks the integrity of its stack **1105**. Next, the value of the reference canary **100** has to be saved, in a designated location **903**, before renewing it at the type 6 function **1101** and restoring it back, before returning from the function **1104** that has checked the frame canary.

4.5 List of references cited

4.5.1 List of Patents

Cited Patent	Filing date	Publication date	Applicant	Title
US 6578094 B1	Mar 2, 2000	Jun 10, 2003	International Business Machines Corporation	Method for preventing buffer overflow attacks
US 6941473 B2	Jan 30, 2001	Sep 6, 2005	International Business Machines Corporation	Memory device, stack protection system, computer system, compiler, stack protection method, storage medium and program transmission apparatus
US 7086088 B2	May 15, 2002	Aug 1, 2006	Nokia, Inc.	Preventing stack buffer overflow attacks
US 7467272 B2	Dec 16, 2004	Dec 16, 2008	International Business Machines Corporation	Write protection of subroutine return addresses
US 7581089 B1	Apr 18, 2007	Aug 25, 2009	The United States Of America As Represented By The Director Of The National Security Agency	Method of protecting a computer stack
US 7660985 B2	Apr 29, 2004	Feb 9, 2010	At&T Corp.	Program security through stack segregation

US 8028341 B2	Oct 27, 2009	Sep 27, 2011	Intel Corporation	Providing extended memory protection
US 8631248 B2	Oct 31, 2007	Jan 14, 2014	Apple Inc.	Pointguard: method and system for protecting programs against pointer corruption attacks

4.5.2 List of Patent Applications

Cited Patent	Filing date	Publication date	Applicant	Title
US 2003/0177328 A1	Mar 13, 2003	Sep 18, 2003	FUJITSU LIMITED	Method and apparatus for controlling stack area in memory space
US 2004/0168078 A1	Dec 2, 2003	Aug 26, 2004	Brodley Carla E., Vijaykumar Terani N., Hilmi Ozdoganoglu, Kuperman Benjamin A.	Apparatus, system and method for protecting function return address
US 2013/0219373 A1	Feb 21, 2013	Aug 22, 2013	International Business Machines Corporation	Stack overflow protection device, method, and related compiler and computing device

4.6 Claims

What Is Claimed:

1. A method for protecting the SSP technique in a program against leaking information regarding the value of the canary of said SSP technique. The method consists of renewing the reference canary at key places in the code while the program is running. The method comprises the following steps:
   (a) Prior to compilation, identify a function, or block of code, such that the already stacked frame canaries are never checked afterwards.
   (b) For each of said identified functions, or blocks of code, it is possible to renew the reference canary. Comprising the following steps:
      i. Compute a new random number,
      ii. Overwrite the reference canary with said new random number.
2. The **claim 1** method, whereby:
   (a) Said identified function, or block of code, does not return and
   (b) Said identified function, or block of code, does not have nested functions which make non-local jumps to a parent function of said identified function or block of code.
3. The **claim 1** method, whereby:
   (a) Said identified function, or block of code, does not return and
   (b) Said identified function, or block of code, does have nested functions which make non-local jumps to a parent function of said identified function, or block of code, and
   (c) Neither the said parent function or block of code, nor none of its return-reachable parent functions, or blocks of code, check the canary value.
4. The **claim 1** method, whereby:
   (a) Said identified function, or block of code, may return and
   (b) Said identified function, or block of code, does not check the canary value and

   (c) None of the return-reachable parent functions, or blocks of code, checks the canary value.

5. The **claim 1** method, involving renewing the reference canary, whereby said identified function, or block of code, is one of the initialising functions of a new thread, task or process (fork, clone or equivalent).

6. The **claim 1** method, involving renewing the reference canary, whereby said identified function, or block of code, is a system call service routine.

7. The **claim 1** method, involving renewing the reference canary, whereby said identified function, or block of code, is the main loop of a worker server.

8. The **claim 1** method, involving renewing the reference canary whereby said identified function, or block of code, which may be exposed to attacks, including:

   (a) Functions that handle user accessible data or

   (b) Functions that use libraries or code from non-trusted sources or

   (c) Functions that start the execution or interpretation or emulation of code that is loaded as plugins.

9. The **claim 1** method, involving renewing the reference canary whereby said identified function, or block of code, which has an exception handler code. Typically, these functions contain a 'try-catch' block of code or save the stack context/environment from doing non-local jumps.

10. A method for protecting the SSP technique in a program against leaking information regarding the value of the canary of said SSP technique. The method consists of renewing and restoring the reference canary at key places in the code while the program is running. The method comprises the following steps:

    (a) Prior to compilation, identify a function, or block of code, such that the already stacked frame canaries are checked afterwards.

    (b) For each of said identified functions, or blocks of code, it is possible to renew and restore the reference canary. Comprising the following steps:

        i. Prior to renewing the reference-canary, store the current reference canary value into a designated location;

        ii. Compute a new random number;

iii. Overwrite the reference canary with said new random number;

iv. Before checking the canary value, the reference canary must be restored, by copying the previously stored reference canary value back to the reference canary.

11. The **claim 10** method, whereby:

    (a) Said identified function, or block of code, may return and

    (b) Said identified function, or block of code, checks the canary value.

12. The **claim 10** method, whereby:

    (a) Said identified function, or block of code, may return;

    (b) Said identified function, or block of code, does not check the canary value and

    (c) At least one return-reachable parent function, or block of code, of said identified function, or block or code, checks the canary value.

13. The **claim 10** method whereby:

    (a) Said identified function, or block of code, does not return;

    (b) Said identified function, or block of code, has at least a nested function which makes a non-local jump to a parent function of said identified function, and

    (c) Said parent function, or block of code, or at least one of its return-reachable parent functions, or blocks of code, checks the canary value.

14. The **claim 10** method, involving renewing the reference canary whereby said identified function, or block of code, which has an exception handler code. Typically, functions that contain a 'try-catch' block of code or save the stack context/environment for doing non-local jumps.

15. The **claim 10** method, involving renewing the reference canary whereby said identified function, or block of code, is the main loop of a worker server.

16. The **claim 10** method, involving renewing the reference canary whereby said identified function, or block of code, which may be exposed to attacks, including:

    (a) Functions that handle user accessible data;

(b) Functions that use libraries or code from non-trusted sources or

(c) Functions that start the execution, interpretation or emulation of code that is loaded as plugins.

Chapter 5

On the effectiveness of NX, SSP, RenewSSP and ASLR against stack buffer overflows

This chapter assesses the effectiveness of the three more common protection mitigation techniques: Non-eXecutable, stack-smashing protector and address space layout randomisation.

The study indicates that the SSP technique is the most effective against stack buffer overflows.

On forking servers, the ASLR technique is almost useless on 32-bit architectures, due to the limited entropy provided by the size of the address space. This finding is the starting point for making improvements to the ASLR in the following part of the thesis.

Contents

5.1 Introduction

Over the last few years, a set of defensive techniques has been developed to protect against malicious users. The security field is a very active, constantly changing area, in which innovations and advances render technology obsolete very quickly. Therefore, it is mandatory to periodically reassess the effectiveness of these techniques. The requisites and constraints that were previously considered when a technique was initially developed may be no longer valid when applied to current systems, and some protection techniques become outdated as a result of changes to the execution framework or newly developed counter-attacks.

Stack-smashing protection [17], the renew stack-smashing protector [24], address-space layout randomisation [31] and Non-eXecutable [32] are techniques that mitigate the execution of malicious code. They do not remove the underlying error which leads to vulnerability, but they do prevent or hinder exploitation of the fault. The key idea behind the first three techniques (SSP, RenewSSP and ASLR) mentioned above is to introduce a secret that must be known by the attacker in order to bypass it, while he NX technique's method involves restricting the execution capabilities of processes.

In this paper, the authors evaluate the effectiveness of each technique when used both individually and when combined, in different execution environments, by considering different error manifestations and different exploitation techniques with respect to stack buffer overflow vulnerability. According to SANS [33], it is the third most dangerous vulnerability in current systems.

The main contributions of this paper are as follows:

1. A statistical characterisation of remote attacks launched against the NX, SSP, RenewSSP and ASLR protection techniques and when used in combination.

2. A detailed analysis of the time needed to break-in, and the probability of success.

3. We identify the scenarios (executing framework, operating system, etc.) which jeopardise the expected effectiveness of the classical techniques (SSP and ASLR), due to information leaks.

4. The results show that the RenewSSP is a promising modification of the SSP which makes the SSP robust against brute force attacks in all scenarios.

This paper is organised as follows. Section 5.2 presents the background and context involved in undertaking the statistical analysis: *i*) The analysed vulnerability; *ii*) The execution environment in which the programs are executed; *iii*) The protection techniques under study; *iv*) Threats to bypassing the protection techniques and *v*) The generic structure of an attack. Section 5.3 presents a statistical analysis of the attacks, in which special attention is given to the forking server, since it is the most widely used. Section 5.4 evaluates the practical effectiveness of the techniques on current systems. Finally, the concluding section summarises the contributions of the paper and outlines the main findings.

5.2 Background and Terminology

The four techniques analysed herein are used with minor modifications in most modern operating systems. Each operating system has its own particularities. In order to avoid excessive duplication, we have used only the UNIX style (`fork()`,`exec()`, etc.) to refer to the way processes are created. The conclusions, however, are applicable to the other systems.

5.2.1 Stack buffer overflow vulnerability

Stack buffer overflow vulnerability (also known as stack-smashing) occurs when a program writes to a memory address on the program's call stack outside of the intended data structure – usually a fixed length buffer. Stack buffer overflow bugs are caused when a program writes more data to a buffer located on the stack than is actually allocated for that buffer, which most of the times results in the corruption of adjacent data on the stack. When the overflow is done accidentally (i.e. it is not malicious), the program behaves improperly, due to data corruption, or executes illegal instructions which trigger a program crash.

However, if the attacker is able to control the way the overflow is produced (i.e. it is intentional), then it may take control of the execution flow of the buggy program in such a way that they may execute arbitrary code. This is illustrated in the listing 5.1, an example including `memcpy()` which shows a trivial example of a stack buffer overflow.

The canonical method for exploiting a stack-based buffer overflow is to overwrite the return address stored in the stack with a pointer to an attacker's selected direction.

```
void vuln_function(char *srcbuff, int lsrcbuff) {
    char buff[48];
    ...
    memcpy(buff, srcbuff, lsrcbuff);
    ...
}
```

LISTING 5.1: Example function which has a stack buffer overflow.

5.2.2 Types of server architecture

The execution environment and architecture of the server have an important impact on the effectiveness of each technique. Attending to the impact of the protection techniques, we have identified three different server architectures.

Single process: A single process server is a program that attends to all client requests. Attending to the internal architecture of the server, we can distinguish three different sub-types: *i*) Sequential, *ii*) Event-based and *iii*) Multi-thread. From the point of view of security, all three sub-types exhibit the same behaviour. We assume that the server crashes when an incorrect, fake request is received, and then the service is stopped at once. There is little chance of breaking into the server, but it is easy to perform a DoS attack.

Inetd: Every client request is attended to by a different process launched from the server using the sequence `fork()`→`exec()`. A new process image is loaded in memory, and so all the secrets used by the child process during each client request are renewed.

We decided to use the inetd name, to honour the old network server daemon. This sequence is also called *self-re-execution*, which is used by the SSH suite.

Forking: The operation of a forking server is very close to that of the inetd, but the child processes are in charge of directly attending to client requests, that is, no new executable image is loaded using the `exec()` call. Therefore, all the children have the same secrets as the parent (except when the new RenewSSP technique is used). The behaviour of these kinds of servers can be used by attackers to perform more effective attacks.

Android applications belong to this ‘category’. All Android applications are child processes of the *Zygote* process. The difference with respect to a conventional forking server is that although each child executes the same Dalvik virtual machine, the application is different on each one.

5.2.3 Protection techniques

Following is an overview of the four techniques analysed in this paper.

NX or DEP: Memory sections (pages) of the process, which contain code, are marked as executable and read-only. On the other hand, those areas containing data are marked as read/write and non-executable. Processors must provide hardware support to check for this policy when fetching instructions from main memory. Even if an attacker successfully injects code into a writeable (not executable) memory region, any attempt to execute this code will lead to a process crash. This technique is also known as 'W^X', because a memory page can be marked as executable or writeable, but not both at the same time.

SSP: A random value, commonly known as a *canary* or a *guard*, is placed on the stack, just below the saved registers, by the function prologue code. This value is checked at the end of the function, before returning, and the program aborts if the stored canary does not match its initial value. Any attempt to overwrite the saved return address on the stack will also overwrite the canary, which leads to a process crash to prevent any intrusion.

ASLR: Whenever a new process is loaded in main memory, the operating system loads the different areas of the process (code, data, heap, stack, etc.) at random positions in the virtual memory space of the process. Attacks relying on precisely knowing the absolute address of the injected code or a library function, such as `ret2libc`, are very likely to crash the process (unless they know the memory map of the target process), thus preventing a successful intrusion.

RenewSSP: This modification of the stack-smashing protector (SSP) technique renews the value of the reference canary of a process on any 'non-returning' function [24]. It is especially effective when used on the child's code, right after the new process is created with the `fork()` or `clone()` calls.

5.2.4 Threats to protection techniques

Over the last few years, several strategies to bypass each protection technique have been developed [20, 34, 35]. Due to space limitations, only the core of each attack strategy is presented herein. We do not consider attacks based on information leaks other than the one that can be obtained from stack buffer overflow vulnerability; other forms of information leak require the existence of additional vulnerabilities and are beyond the scope of this work.

Following is a brief description of the attacks considered in this paper.

NX/DEP: The Non-eXecutable bit (NX)/data execution prevention (DEP) mechanism can be bypassed by using attacks that do not need to execute the injected code but reuse the already existing and mapped code on the target application. There is a family of techniques referred to as *ret2** [36] and more generally the return-oriented programming (ROP) technique [37]. ROP is a very effective technique for bypassing the NX. As a result, an ASLR counter-measure was developed.

It is realistic to assume that modern attacks do not inject code but use the ROP method. Therefore, from now on, we will assume that NX bit protection is bypassed directly, following which security relies on the effectiveness of the remaining techniques. It is important to point out that although the NX is defeated by the ROP, it must not be considered deprecated, and shall be maintained as far as the ASLR is not 100% secure and the NX does not introduce any execution overhead.

SSP-tat: SSP trial-and-test. If the canary value is replaced or renewed after each trial, then the experiment is known as ‘sampling with replacement’, whereby the attacker can try at will, but it cannot discard the already tested value.

SSP-bff: SSP brute force full. In order to perform this attack the target service always has the same canary value and the service is restarted automatically after any server crash. The attacker can try as many times as required during the attack. On each trial, the attacker guesses a different value of the canary until it matches. Since the canary value is always the same on the server, the attacker can discard incorrectly guessed values. Statistically, this is known as the ‘sampling without replacement’ experiment. Typically, values are tested sequentially, starting from zero and then rising up to the maximum value.

SSP-bfb: SSP byte-for-byte. If the manifestation of the error allows attackers to overflow up to the desired byte with any value, then it is possible to perform a byte-for-byte attack, which involves trying all possible values for each byte sequentially. The code on listing 5.3 implements the code of the attack [12],[38]. All values from zero to 255

```
for (k=0; k<c; k++)
  if( OK == end_request_up_to_canary(k) )
    break;
printf("Canary value:  %d\n", k);
```

Listing 5.2: Brute force to the canary.

are tried sequentially until the correct value is found. The process continues with the next byte until all the bytes have been found.

This method allows one to build very fast attacks. Unfortunately (for the attacker) this possibility is quite odd.

RenewSSP-tat: RenewSSP trial-and-test. A brute force attack cannot be employed against the RenewSSP, as pointed out by the authors in [24]. In this scenario the only attack strategy is trial-and-test against the whole canary, independently of the type of server (single, inetd or forking).

ASLR-bff: ASLR brute force full. To bypass the ASLR, the attacker needs to know the absolute address at which the ROP 'program' starts [39]. If the memory map is the same in all the attacker trials, and the attacker can perform as many trials as required, then it is possible to build a brute force attack [35], that is, an experiment 'without replacement'.

ASLR-tat: ASLR trial-and-test. When the memory map of the server is renewed after every trial (of after a failed trial), then the attacker

```
union {
    unsigned char single_bytes[n];
    unsigned int  full_val;
} secret;
int idx, k=0;
for (idx=0; idx<n; idx++){
    for (a=0; a < 256; a++){
        if ( OK == send_request_up_to(idx, a) ){
            secret.single_bytes[idx]=a;
            k += (a + 1);
            break;
        }
    }
}
printf("Secret value: %x\n", secret.full_val);
printf("Trials needed: %d\n", k);
```

Listing 5.3: Byte-for-byte attack.

can try different base address values until one matches. However, it cannot discard already tested ones.

ASLR-one: ASLR one-shot. If there are one or more memory sections of the server that are not randomised, attackers can use static (and therefore known) areas to build the ROP sequence. For example, when the code of the application (not the libraries) is not randomised, then it is possible to build a one-shot attack with a probability of 95.6% in x86, and 61.8% in x86_64, as shown by Roglia et al. [34]. This attack is effective in all server architectures.

Another strategy employed to bypass the ASLR is by directly observing the memory map of the target. Most operating systems' (Windows, Android applications and the other major player OSs) libraries are randomised only per boot time and shared between all applications. Any local user knows the ASLR secret. On these systems, the ASLR is completely useless against a local attack.

GNU/Linux, which implements the position-independent code (PIC) for libraries and when the executable is compiled with position-independent executable (PIE), is not affected by ASLR-one attacks.

Another form of disclosure is through the information which some applications automatically report to the vendor provider (as debugging information) after a crash, which could contain valuable information that is useful to an attacker.

We will consider that the ASLR is bypassed with a ASLR-one when there are enough gadgets to build the attack, according to [34], or when it is a local attack on systems where ASLR randomisation is only done at boot time.

5.2.5 Generic structure of an attack

In this work, we consider that the attackers have access to the following information: *i*) Source code, *ii*) Compiler and built options and *iii*) The execution environment of the target. The attacker can work off-line, using an in-house replicated target, by testing and tuning the attack as long as necessary before it actually starts.

The work that can be done off-line is considered to have no cost. That is, it takes zero time to achieve. This is a realistic assumption (from the defendant's point of view), because the attack starts only when the server is effectively attacked.

The attack consists of sending fake client requests, specially designed to overflow a buffer. The faked client request can be seen as a string long enough to flood the buffer with the following elements (figure 5.1):

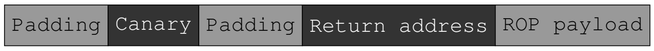


FIGURE 5.1: Fields of a fake request.

Padding the canary: Extra bytes are inserted, to increase the length of the request. The number of added bytes must be computed so that the next field overwrites the stack canary exactly. The length of this field can be accurately estimated off-line from the binary image of the server and a few trials against the target server. Since the cost of this part is relatively low, we will assume that the attacker knows this value.

Canary: This field will overwrite the canary. In order to succeed, it is necessary to know the actual value of the frame canary used in the target server. The canary is commonly a word (4 or 8 bytes). Let C be the entropy bits of the canary.

Padding the return: Typically it is a few bytes (4 or 8 depending on the platform). We will assume that the attackers do not need to know this value (to build an ROP).

Return address: Absolute entry point of the ROP code. The ROP code is located in a section with execution rights. We will suppose that the attacker must know the current memory layout of the server. Let R be the entropy bits of the ASLR.

ROP payload: The injected ROP payload. This payload is basically a list of gadget addresses. Gadgets are blocks of code located in relative positions with respect to the ROP entry point. Therefore, once the attacker knows the entry point of the ROP, the rest of the ROP payload can be automatically adjusted with the appropriate offset. We will assume that the attackers are able to both build the ROP and adjust the resulting payload to the server memory layout, once the entry point is known (i.e. the jump address). Therefore, no extra information is required to build this field.

The grey fields of the request on Figure 5.1 can be filled by the attacker inspecting server code off-line, though dark fields can only be obtained through direct 'interaction' with the target. In most cases the attacker has a very limited and controlled interaction path.

It can only submit a faked request and wait for the result. There are basically only two possible values for the result:

Symbol	Description
C	entropy bits of the canary.
n	number of entropy bytes of the canary ($n = C/8$).
c	number of values that can take the canary ($c = 2^C$).
R	entropy bits of the ASLR for libraries.
r	number of places where the library can be located ($r = 2^R$).
k	number of trials carried out by an attacker on a service.

Table 5.1: Summary of symbols.

1. The server returns an answer which is interpreted by the attacker as a correct guess. That is, the fake request does not crash the server.
2. The server does not respond, which is interpreted as an incorrect guess. The server should have crashed.

Depending on the architecture and execution environment of the server, they interpret the result differently (success or failure) and will tune or adjust the fake request.

5.3 Analysis of the protection techniques

This section analyses the protection mechanisms for each server type. The probability of a successful attack is measured as the number of trials needed to break into the system.

5.3.1 Single process server

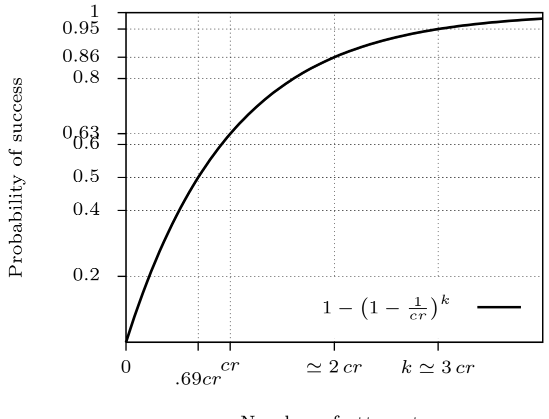
The attacker has only one single trial (assuming that the server service is not restarted by the administrator) on both the SSP and the ASLR. The probability of breaking into the system is given by the Bernoulli distribution:

$$Pr(\mathcal{X} = n) = \begin{cases} 1 - \frac{1}{cr} & \text{if } n = 0, \text{"failure"} \\ \frac{1}{cr} & \text{if } n = 1, \text{"success"} \end{cases} \tag{5.1}$$

As the values of c and r are commonly large, there is little chance to break in. Moreover, there is little interest in trying to break in unless other vulnerabilities or info leaks are available (a notion which is beyond the scope of this paper). This type of server has been included for completeness.

5.3.2 Inetd-based server

The attacker can carry out as many trials as needed. On each trial, they have the same probability of success: $\frac{1}{cr}$. There is no benefit to attacking first the canary and then the return address, because there is no way to learn from previous failures. The attack is SSP-tat jointly with the ASLR-tat.



Geometric	
PMF	$\frac{1}{cr}\left(1-\frac{1}{cr}\right)^{k-1}$
CDF	$1-\left(1-\frac{1}{cr}\right)^{k}$
Mean	$\mu = cr$
Variance	$\sigma^2 = \frac{1-cr}{cr}$
Trials for 100%	$= \infty$
95%	$\simeq 3\,cr$
50%	$\simeq 0.693\,cr$

TABLE 5.2: Inetd-based server summary.

This strategy is modelled as a Bernoulli trial experiment, in which k trials are carried out with a success probability of $\frac{1}{cr}$ in any trial. We are interested in counting the number of trials needed to get the first success, which follows a geometric distribution defined for an infinite number of trials in the range $k \in [1, \infty[$. The probability that the k^{th} trial will be the first success is given by the PMF:

$$Pr(\mathcal{X} = k) = \frac{1}{cr}\left(1-\frac{1}{cr}\right)^{k-1} \tag{5.2}$$

The cumulative distribution function (CDF) provides more valuable information; rather than the probability of succeeding at exactly the k^{th} trial, we are interested in the probability of succeeding at any time up to the k^{th} trial. The CDF is defined as $Pr(\mathcal{X} \leq k)$ and is given by:

$$Pr(\mathcal{X} \leq k) = \sum_{i=1}^{k} \frac{1}{cr}\left(1 - \frac{1}{cr}\right)^{i-1} = 1 - \left(1 - \frac{1}{cr}\right)^{k} \tag{5.3}$$

Since both secrets must be correctly guessed at once, the probability of success at each trial is one out of $c \times r$.

5.3.3 Forking server

The behaviour of these kinds of servers can be used by attackers to perform more effective assaults. The rest of this section covers in detail how each protection technique can be bypassed, both individually and when used in combination.

5.3.3.1 SSP brute force full (SSP-bff)

It is assumed that the behaviour (success or failure) of the server can be detected; for example, an incorrect guess closes the connection abruptly.

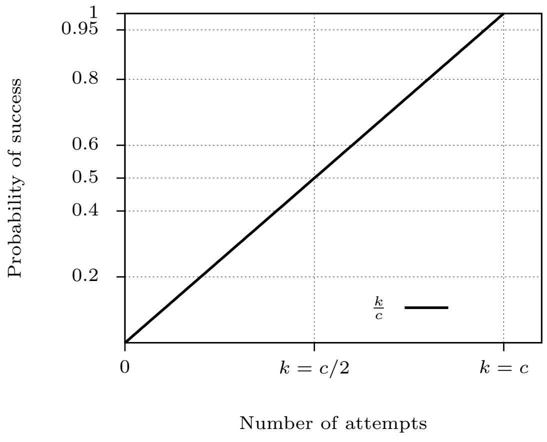
The probability that on the k^{th} trial the attacker will try the correct value is given by uniform distribution, with a PMF given by $Pr(\mathcal{X}_c = k) = \frac{1}{c}$. Additionally, the cumulative distribution function (CDF) is the sum of the PMF: $Pr(\mathcal{X}_c \leq k) = \sum_{i=1}^{k} \frac{1}{c} = \frac{k}{c}$. This distribution function is only 'valid'[1] in the range $[0, k]$. Table 5.3 is a summary of the attack against the whole canary.

5.3.3.2 SSP byte-for-byte (SSP-bfb)

Note that overflows caused by most string manipulation functions cannot be used to implement this attack, because a null byte is always copied at the end.

Each brute force attack against a single byte can be modelled as a uniform distribution. The sum of several uniform distributions, n in our case, is known as the 'Irwin-Hall distribution', which quickly (for $n > 3$) approximates – quite accurately for our purposes – to a normal distribution. The figure in Table 5.4 shows how the CDF changes in line with

[1]To be mathematically correct it should be said that its "*support is*".



Uniform	
Mean	$\mu = c/2$
Variance	$\sigma^2 = (c-1)/12$
PMF	$1/c$
CDF	k/c
Trials for 100%	$= c$
95%	$= 0.95c$
50%	$= c/2$

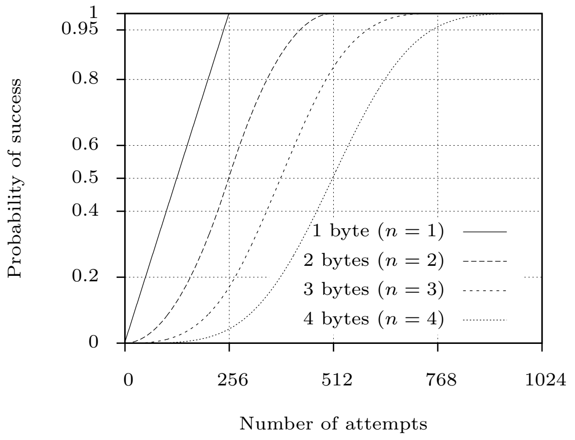
TABLE 5.3: Summary of the SSP-bff.

the length (number of bytes) of the canary. It is important to note that regardless of the number of bytes, all CDFs reach the value of one when $a = 256 \times$ BYTES_PER_WORD. That is, in the worst case, they have to carry out $256 \times$ BYTES_PER_WORD trials to break the canary.

A vulnerability of this type is very dangerous, as the canary can be obtained in no more than 1 second, regardless of the word width of the architecture.

5.3.3.3 RenewSSP trial-and-test (RenewSSP-tat)

This attack strategy is modelled as a Bernoulli trial experiment, whereby k trials are executed with a success probability of $\frac{1}{c}$ in any trial, while the number of trials needed to break into the system is modelled as a geometric



Sum of n uniforms $\simeq$ Normal when $n > 3$	
Mean	$\mu = \frac{256n}{2} = \frac{256\log_2(c)}{2}$
Variance	$\sigma^2 = \frac{(256-1)n}{12}$
PMF	$\simeq \frac{1}{\sqrt{2\pi\sigma^2}} e^{\left(-(x-\mu)^2/2\sigma^2\right)}$
CDF	$\simeq \frac{1}{2}\left(1 - erf\left(\frac{k-\mu}{\sqrt{2\sigma^2}}\right)\right)$
Trials for 100%	$= 2\mu$
95%	$= \mu + 1.645\sigma^2$
50%	$= \mu$

TABLE 5.4: Summary of the SSP-bfb.

distribution. The summary is in Table 5.2, where the value of $r = 1$, since in this case we are considering only bypassing the canary.

5.3.3.4 ASLR brute force full (ASLR-bff)

In forking servers, library mapping is inherited by all children. Therefore, the ASLR-bff exhibits the same behaviour (distribution) as the attack on the whole canary, i.e. a uniform distribution, by sampling without replacement. The mean is $r/2$ and its range is $[0, r]$. Table 5.3 can be applied to the ASLR-bff attack, by only changing the variable c by r.

5.3.3.5 SSP brute force full + ASLR brute force full (SSP-bff + ASLR-bff)

Since it is possible to split the attack into two phases, we first attack the canary and then the ASLR. When the whole word of the canary has to be the attacked, the resulting distribution is the sum of two uniforms, where each uniform has a different range of values: $[0, c]$ for the canary and $[0, r]$ for the ASLR. The sum of two different uniforms gives a trapezoidal distribution. If $c = r$, it becomes triangular. For simplicity, we will assume that $c > r$. The PMF is given by:

$$Pr(\mathcal{X} = k) = \begin{cases} \frac{k-2}{(c-1)(r-1)} & \text{for } k \in [2, j+1[\\ \frac{1}{c-1} & \text{for } k \in [r+1, c+1[\\ \frac{c+r-k}{(c-1)(r-1)} & \text{for } k \in [c+1, c+r[\end{cases} \tag{5.4}$$

When the value of r is much smaller than that of c, as is the case on real systems, the expression 5.4 can be approximated to a uniform distribution.

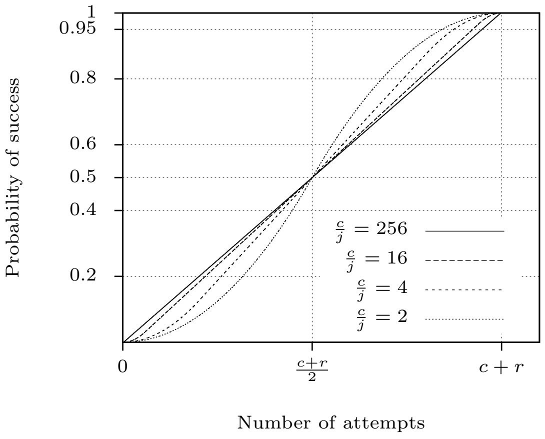
5.3.3.6 SSP byte-for-byte + ASLR brute-force-full (SSP-bfb + ASLR-bff)

In this case it is also possible to split the attack to bypass first the SSP and then the ASLR. The statistical distribution of the attack on the canary, plus the ASLR, is given by the sum of the distributions of both random variables. If the canary can be attacked with the SSP-bfb method, then it can be computed as the sum of $n + 1$ uniform variables (n of the SSP plus the uniform of the ASLR-bff), where n is the number of unknown bytes of the canary and $R/8$ is entropy bytes introduced by the ASLR. The result is even closer to a normal distribution, the parameters for which are:

$$\begin{array}{rl} \text{Mean} & \mu = \frac{256(n+R/8)}{2} \\ \text{Variance} & \sigma^2 = \frac{(256-1)(n+R/8)}{12} \end{array}$$

5.3.3.7 RenewSSP trial-and-test + ASLR trial-and-test (Renew-SSP-tat + ASLR-tat)

The attack strategy employed to bypass this combination of techniques is similar to that used for the inetd server. There is no benefit to attacking the canary first and then the ASLR, because there is no way to learn from/discard previous trials. Each trial has the same probability of success: $\frac{1}{cr}$. This strategy is modelled as a Bernoulli trial experiment, in which k



Trapezoidal

$\simeq$ Uniform when $c/r > 256$

Mean	$\mu = \frac{c+r}{2}$
Variance	$\sigma^2 = \frac{c+r-2}{12}$
PMF	$\simeq \frac{1}{c+r}$
CDF	$\simeq \frac{k}{c+r}$

Trials for 100%	$= c + r$
95%	$= 0.95\,(c + r)$
50%	$= 0.50\,(c + r)$

Table 5.5: Summary of the SSP + ASLR full attack.

trials are carried out with a success probability of $\frac{1}{cr}$ in any trial. Equation 5.2 shows the probability mass function (PMF), while the cumulative distribution function (CDF) is shown in equation 5.3.

5.3.4 Server summary

This section is a summary of the most relevant statistical parameters for single, inetd and forking servers. The forking server is the most interesting here, because it is widely used in real systems.

In real systems, NX, SSP or RenewSSP and ASLR are used simultaneously. Table 5.6 shows the distribution, mean, variance and trials required to break the system with a probability of 100%, 95% and 50%. The same

Table 5.6, with the value of $r = 1$, represents the cost of the attacks when the ASLR can be bypassed with the ASLR-one attack.

			Trials for a prob. of:		
	Distrib.	μ	100%	95%	50%
Single: Single-Shoot	Bernoulli	$\frac{1}{cr}$	–	–	–
Inetd: SSP-tat+ASLR-tat	Geom.	cr	∞	3μ	0.693μ
Forking: SSP-bff+ASLR-bff	Uniform	$\frac{c+r}{2}$	2μ	0.95μ	μ
Forking: SSP-bfb+ASLR-bff	Normal	$\frac{2^8 n+r}{2}$	2μ	$\mu + 1.645\sigma^2$	μ
Forking: RenewSSP-tat+ ASLR-tat	Geom.	cr	∞	3μ	0.693μ
Forking: RenewSSP-tat+ ASLR-one	Geom.	c	∞	3μ	0.693μ

TABLE 5.6: Summary of the most common systems and attacks.

5.4 Discussion

In order to evaluate the effectiveness of the NX, SSP, RenewSSP and ASLR, we selected the most common server architectures and configurations. Current systems are all protected by the three protection techniques NX, SSP and ASLR. We also included the RenewSSP technique, which, although not widely used currently, we expect will replace the original SSP in the near future.

The cost is measured as the number of attempts (trials) needed by the attackers to break into the system, summarised in Table 5.7.

The system is broken when the secrets are correctly guessed. The values are calculated using the following parameters:

- The system configuration (processor, network, firewalls, etc.) allows the attacker to perform 1000 trials per second.

	Attack/Bypass	100%	Mean
32bits syst.	SSP-bff + ASLR-bff	4 Hours	2 Hours
	SSP-bff + ASLR-one	4 Hours	2 Hours
	SSP-bfb + ASLR-bff	1 sec	< 1 sec
	SSP-bfb + ASLR-one	< 1 sec	< 1 sec
	RenewSSP-tat + ASLR-one	∞	3 Hours
	RenewSSP-tat + ASLR-tat	∞	34 Days
64bits syst.	SSP-bff + ASLR-bff	2.32 Myr	1.16 Myr
	SSP-bff + ASLR-one	2.32 Myr	1.16 Myr
	SSP-bfb + ASLR-bff	74 Hours	37 Hours
	SSP-bfb + ASLR-one	1 sec	< 1 sec
	RenewSSP-tat + ASLR-one	∞	1605.79 Kyr
	RenewSSP-tat+ASLR-tat	∞	431.05 Tyr

TABLE 5.7: Time cost for attacks in forking servers at 1000 trials/sec.

- The entropy of the SSP and RenewSSP is 24 and 56 bits for 32-bit and 64-bit systems, respectively.
- The entropy of the ASLR is 8 and 28 bits for 32-bit and 64-bit systems, respectively.

On systems which are regularly monitored by humans or by advanced event correlation tools, the techniques are effective if the time required to bypass them is longer than the reaction time. Protection for a few hours can give defenders enough time to apply specific corrective measures. On stand-alone, non-supervised systems, the system should resist in the order of years, to be considered effective.

Table 5.8 is a more complete list of systems and attacks, including combinations that are no longer released but may be still operative.

The following list summarises the most important results of this evaluation.

- The NX was rendered mainly obsolete, first by the family of ret2* attacks and then by the ROP. Although it slightly increases the difficulty of building an exploit, since it is an un-expensive technique (the check is performed by hardware: the MMU), it is still worthwhile using it. Basically, there is no benefit in removing it from a system where it is already implemented.
- In the inetd architecture, the combination of the three techniques (NX, SSP and ASLR) is very effective, as it has a multiplicative effect.

  This robust architecture is used by the SSH suite. Each connection request is handled with the following sequence of system calls: `fork()` $\rightarrow$

		32 bits		64 bits	
	Technique	**100%**	**Mean**	**100%**	**Mean**
Inetd based	SSP-tat	∞	1.2×10^7	∞	5.0×10^{16}
	ASLR-one	1×10^0	0.5×10^0	1×10^0	0.5×10^0
	ASLR-bff	∞	1.8×10^2	∞	1.9×10^8
	SSP-tat+ASLR-one	∞	1.2×10^7	∞	5.0×10^{16}
	SSP-tat+ASLR-tat	∞	3.0×10^9	∞	1.3×10^{25}
Forking based	SSP-bff	1.7×10^7	8.4×10^6	7.2×10^{16}	3.6×10^{16}
	SSP-bfb	7.7×10^2	3.8×10^2	1.8×10^3	9.0×10^2
	RenewSSP-tat	∞	1.2×10^7	∞	5.0×10^{16}
	ASLR-one	1×10^0	1×10^0	1×10^0	1×10^0
	ASLR-bff	2.6×10^2	1.3×10^2	2.7×10^8	1.3×10^8
	SSP-full+ASLR-one	1.7×10^7	8.4×10^6	7.2×10^{16}	3.6×10^{16}
	SSP-bff+ASLR-bff	1.7×10^7	8.4×10^6	7.2×10^{16}	3.6×10^{16}
	SSP-bfb+ASLR-one	7.7×10^2	3.8×10^2	1.8×10^3	9.0×10^2
	SSP-bfb+ASLR-bff	1.0×10^3	5.1×10^2	2.7×10^8	1.3×10^8
	RenewSSP-tat+ASLR-one	∞	1.2×10^7	∞	5.0×10^{16}
	RenewSSP-tat+ASLR-tat	∞	3.0×10^9	∞	1.3×10^{25}

TABLE 5.8: Attempts to bypass the protection techniques.

`exec(sshd)`$\rightarrow$ `do_work`$\rightarrow$ `exit()`. This way, it receives all of the randomisations the operating system can provide.

In 64-bit systems it is impossible to bypass, as the mean time is at least 1,605,000 years with a probability of 95% in the best case for the attacker. In 32-bit systems, although less effective, it is still provides acceptable protection.

This architecture is not affected by brute force attacks. The dangerous byte-for-byte type of attack cannot be employed by attackers, even when vulnerability allows one to overwrite at byte granularity.

- Unfortunately, the forking server architecture greatly reduces the effectiveness of the protection, because it allows new exploitation strategies:
  - Split the attack of the SSP and the ASLR, which has an additive effect rather than a multiplicative one.
  - It is possible to launch brute force attacks on both the SSP and the ASLR secrets.
  - If the manifestation of the vulnerability allows overwriting at the byte level, it is possible to make byte-for-byte attacks.

In forking servers, the protection techniques are only effective in 64-bit systems when byte-for-byte cannot be employed. The byte-for-byte attack renders useless the SSP, and the ASLR is not strong enough to counter an assault on its own.

The RenewSSP technique restores the effectiveness of both the SSP and the ASLR in forking servers. The attacker can no longer discard tested values (i.e. brute force attacks cannot be made against the SSP). Also, it is not possible to split the SSP-ASLR attack. Therefore, the RenewSSP technique provides the same level of protection as that offered by the inetd.

- Both SSP and the ASLR techniques are basically useless against local attacks for Android applications, as all Android applications share the same canary and memory maps.

  This problem only affects Android applications, that is, those that use the Dalvik virtual machine. On the other hand, native Android processes enjoy one of the best sources of protection available[2].

- The ASLR on Windows and bitten fruit OSs is implemented on a per-boot basis, which greatly reduces effectiveness against local attacks, as the attacker knows the layout of all the libraries.

5.5 Conclusions

In this paper the authors have reassessed the effectiveness of three mature techniques, namely NX, SSP and ASLR, as well as the new RenewSSP (strictly speaking, it cannot be considered a new technique but rather an improvement to the standard SSP). The study has focused on stack buffer overflow vulnerabilities, which is still one of the most serious security issues in present-day computing.

Besides the direct exploitation of the buffer overflow, this paper considers the presence of multiple attack vectors, for example the possibility of obtaining information from the target service, by using applications that collect public data from within the same system. Another attack vector considered is the weak implementation of the ASLR on most systems (Windows, Android and others) resulting in the fact that all the applications share the same library maps, which renders the ASLR useless in relation to local attacks.

Although the NX/DEP was a revolutionary technique when initially developed, currently it has been rendered obsolete by new attack methods, namely ret* and ROP. Our evaluation indicates that the ASLR on 64-bit systems is an effective counter-measure against these new attacking strategies, but it is not for 32-bit systems. SSP effectiveness is reasonably good

[2]As far as the current published state of the art is concerned.

for both 64-bit systems but it is rather weak for 32-bit architectures even when it is combined with the ASLR.

Both techniques, ASLR and SSP, rely on keeping secret some internal keys (the guard and the memory map for SSP and ASLR, respectively). There is a lot of information shared by and handed down from the parent to its children processes, and the fact that the normal policy applied when a server process crashes is to restart the process automatically, this allows an attacker to make brute force attacks. The more often the protection secrets are renewed, the harder it will be for attackers to bypass them successfully.

Some systems (Android applications, Windows OSs and the bitten fruit company OS) renew ASLR secrets once per boot, while GNU/Linux renew them on a per-process (`exec()`) basis. The once per boot approach is not robust against local attacks.

Our results show that the forking server architecture greatly reduces the effectiveness of protection techniques, especially when the target system is vulnerable to a dangerous byte-for-byte attack. When this type of vulnerability is present in an application, the only solution is the recently proposed extension to SSP, called ‘RenewSSP’.

Part II

Address Space Layout Randomization (ASLR)

Chapter 6

On the Effectiveness of Full-ASLR on 64-bit Linux

We have identified a security weakness in the implementation of the ASLR in GNU/Linux when the executable is PIE-compiled. A PoC attack is described herein, in order to illustrate how this weakness can be exploited. Our attack bypasses the three most widely adopted and effective protection techniques: No-eXecutable bit (NX), address space layout randomisation (ASLR) and stack-smashing protector (SSP). A remote shell is created in less than one second.

Finally, after analysing different mitigation alternatives, we conclude that a new ASLR design is needed.

Some preliminary ideas about a new ASLR design are also proposed, but the completed and detailed analysis and its design are presented in the next chapter.

Contents

6.1 Introduction

Address space layout randomisation (ASLR) is a defensive technique which randomises the memory address of software in an attempt to deter attackers, which relies on knowing the location of an application's memory map. Rather than increasing security by removing vulnerabilities from the system, in the same way that source code analysis tools [40] do, ASLR is a prophylactic technique which tries to make it more difficult to exploit existing vulnerabilities.

The ASLR is commonly complemented by the well-known and widely used stack-smashing Protector (SSP) and No-eXecute (NX)[1] techniques. When these three techniques are properly implemented on a system they provide a strong defence against most memory error exploitation attempts.

Unfortunately, it is not always possible to implement these techniques correctly, and it is beyond the scope of this paper to present an exhaustive list of improper or partial implementations. What follow are therefore just a few illustrative examples. The NX requires hardware support, otherwise it cannot be efficiently implemented; several embedded processors lack NX support, and current attacks do not execute the code injected [11]. ASLR is a simple concept whereby all the addresses that the attackers may use to build an exploit are unknown (hard to guess) to them, but a complete implementation (all sections located in random places) may cause compatibility issues. Another limitation is the reduced range of addresses to allocate the sections, as most 32-bit systems only have 8 bits of effective entropy. The main problem with SSP comes from the small range of random values of the canary on 32-bit systems [24].

A technique that is known to be very effective, but which is very often improperly used, provides a dangerous false sense of security that can be easily exploited by attackers. The fault or weakness remains latent for a long period of time, which then enables the attackers to prepare multiple strategies and tools that will effectively bypass barriers that are generally considered as unbreakable (or properly settled), which is mainly true, except on 'improperly' implemented systems.

The security offered by ASLR is based on several factors [14], including how predictable the random memory layout of a program is, how tolerant an attack technique is to variations in memory layout and how many exploitation attempts an attacker can practically make.

In this paper we analyse the effectiveness of address space layout randomisation in multiple randomised instances of a single application. In particular

[1] Also known as data execution prevention (DEP) or Write XOR eXecute (W ^X).

we implement a new attack based on a stack buffer overflow which can defeat the ASLR in less than 1 second on a machine running 64-bit Linux with full ASLR.

The contributions of this paper are as follows:

1. A weakness disclosure of the ASLR in GNU/Linux.
2. An attack which takes advantage of the weakness successfully bypasses the full ASLR GNU/Linux on a 64-bit system in less than 1 second.
3. Some preliminary ideas about a new ASLR design which is not vulnerable to our attack.
4. A discussion about preventative techniques against our attack.

The rest of this paper is organised as follows: the next section provides an overview of the ASLR technique and the background needed to follow the rest of the paper. In section 6.3 the weakness of the ASLR is presented, and the PoC which exploits it is in section 6.4. Existing countermeasures to mitigate the attacks that can use this weakness are discussed in section 6.5. A new ASLR design is presented in section 6.6, and the paper finishes with the compulsory Conclusion section, in which the main findings and contributions are summarised.

6.2 ASLR Design

The core idea of the ASLR is to place all process sections (data, bss, heap, text, libs, etc.) at random addresses. Rather than 'random addresses', it is probably more accurate to define them as addresses that are unknown and hard to guess.

Address space randomisation hinders some types of security attacks by making it more difficult for an attacker to predict target addresses. For example, attackers trying to execute return-to-libc attacks [41] must know (or compute) the location of the target function. Other attackers trying to execute shellcode injected into the stack, or other writeable and executable sections, have to know the address where the code has been injected. In both cases, the system obscures related memory addresses from the attackers, and these values have to be guessed to bypass the ASLR successfully.

ASLR security is based upon the low probability of an attacker guessing the locations of randomly-placed areas, and so the more entropy, the more secure the system. There are three different entropy dimensions:

1. Non-randomised sections. It is widely accepted that even a single non-randomised section can be used by an attacker to defeat the ASLR. Therefore, all sections must be randomised.
2. Range of entropy. The size or range of possible values where each section can be located. The entropy range used to be different for each type of memory. The larger the range, the better.
3. Relocation frequency. The frequency at which the sections are mapped. Ideally, every process should have a custom memory space where all the sections are located in different places with respect to previous executions of the same executable, and with respect other concurrent processes. The more frequent, the better.

On most systems, the initial implementations of the ASLR used to rely on a shared library infrastructure. Therefore, the ASLR was initially applied only to stack and libraries, which was very effective against direct return-2-lib attacks. Advances in ROP (return-oriented programming) [15] and related techniques have allowed attackers to build programs on almost any section of code that is not randomised, which has stimulated the need for a full implementation of the ASLR. As of the time of writing this paper, there are still a number of systems that do not support full ASLR or treat it as an optional extra.

The range of entropy is seriously limited by available virtual memory space. It is almost impossible to have a 'decent' implementation on 32-bit systems, since with only 256 possible values, it is considered almost useless and a simple brute force attack can defeat the ASLR in a few milliseconds. However, on 64-bit systems, the range is large enough to effectively discourage attackers unless another method to extract information from the target process is available. Even in unrealistic attacks where the system does not provide SSP and NX bit protection [42], the time taken to bypass the ASLR oscillates between 1.7 hours and 34.1 hours.

The last source of entropy comes from the refresh frequency. This feature is directly related to how shared libraries are handled and shared between processes. If the shared libraries must be mapped on the same virtual addresses in all the processes, then ASLR can only be done on a 'per-boot' basis. That is, only the very first time that a library is loaded is it randomly mapped, and so posterior processes must use the library at the already mapped place. This sequence produces a single memory mapping of libraries at the system level which is only renewed when the system reboots.

PaX published the first design and implementation of ASLR [31] in July 2001. The PaX project implementation is the most complete and advanced, also providing kernel stack randomisation from October 2002 onward. It also continues to provide the most entropy for each randomised layout compared to other implementations.

Two years after ASLR was invented and published as part of the Page_EXec (PaX) project, a popular security patch for Linux, OpenBSD became the first mainstream operating system to support partial ASLR (and to activate it by default) [43]. OpenBSD completed its ASLR support after Linux, in 2008, when it added support for PIE binaries [44].

Microsoft® Windows Vista® (released January 2007) was the first version of Windows® operating system to support ASLR [45]. Then all subsequent versions of Windows OS also supported ASLR [46]. There is a wide range of implementations with different levels of entropy, depending on the version and the security configuration: the Enhanced Mitigation Experience Toolkit (EMET), High Entropy ASLR or ForceASLR. For the purpose of this paper, we are only interested in the relative positions where each section of the program is loaded. Since all versions of Windows allocate libraries on a per-boot basis, and the application executable is loaded at a random position with respect to the already loaded libraries, our technique does not apply to Windows.

Apple® first introduced the randomisation of some library offsets in Mac OS X® v10.5 (released October 2007) [47]. However, because this initial implementation was limited to only certain system libraries, it was naturally unable to protect against many attacks that a full ASLR implementation is designed to defeat. In Mac OS X Lion 10.7, Apple expanded its ASLR implementation to also cover application code. Apple stated that “address space layout randomisation (ASLR) has been improved for all applications. It is now available for 32-bit apps (as is heap memory protection), making 64-bit and 32-bit applications more resistant to attack.”

As for OS X Mountain Lion 10.8, the kernel, as well as kexts and zones, are randomly relocated during the system boot. As in the case of Windows, all applications see a concrete library at the same address.

Linux employs the more advanced implementation of ASLR whereby libraries are compiled as position-independent code (PIC), which allows one to share the same executable image among several processes and each process can map the library at different addresses. As a result, ASLR implements ‘per-process’ randomisation. The numbers of bits used to randomise the memory areas vary from one version to another. Default entropy is 28 bits for mmapped areas, while the PaX implementation operates with 35 bits of entropy (see Chapter 7 for more details), which is far more effective against full-word brute force[2] attacks.

Only the code that has been compiled to be relocatable or to be position-independent (PIC or PIE) can be randomised. Typically, only the executables which are more exposed to attacks (such as Web browsers, system commands and the like) are PIE-compiled. By default, application code is compiled to be position-dependent. Several authors suggest that the overheads introduced by PIE are reasonably low compared to the security benefits. Executables compiled as PIE [48] can be loaded to any address, and conversely to normal executables (i.e non-PIE), these kinds of executables can benefit from the ASLR infrastructure. In Linux they are loaded as if they were another dynamic shared library.

6.2.1 PIC & PIE overview

Libraries can be relocated easily thanks to the strong effort made by operating system and compiler designers to reduce application memory footprints by sharing the library code among all running processes. There are two main approaches involved in sharing a library:

1. Load time relocation.
2. Position-independent code (PIC).

The first solution, load time relocation, is the fastest (on i386 and other processors which lack instruction-relative addressing), but it forces one to map the already loaded library in the same virtual addresses in all of the subsequent processes that want to use (share) it. Basically, the first time that a library is loaded, the system allocates a base address for it and links/relocates the code of the library to work at the given addresses. The next application that uses the library must place it at the already assigned virtual directions. That is, all libraries are allocated at random offsets (chosen at boot time), and all the applications share the same offsets. Library code is modified to work on the designated addressed once loaded in RAM. This is a problem on 32-bit systems due to the small range of addresses, but it is not a big issue on 64-bit systems.

A more advanced and flexible solution is to generate code that does not depend on the direction in which it is located but can be executed independently at any position, namely PIC code. This code works with offsets relative to the PC (program counter) rather than absolute addresses, and

[2]'Full-word brute force' refers to the fact that at each trial, the full word is guessed in order to distinguish it from byte-for-byte brute force, where a single byte is guessed on each trial.

it can be loaded once in physical RAM and mapped at any virtual address in each process. On the one hand, unlike in the x86_64, PIC code is slower on the i386 family due to a lack of PC-relative addressing (detailed analysis is beyond the scope of this paper), but on the other hand, code loaded in RAM is exactly the same as code in the library file, which makes swapping slightly more efficient.

PIC libraries can be freely loaded at any address and in any process. Regarding security, the PIC mechanism provides a higher level of entropy, because each process may have a different map. Linux effectively maps each library in a different location for each process.

The last step in randomising application code involves randomising the directions in which the application code is executed. Note that all of the previous mechanisms are the consequence of library sharing efforts. However, application code, which is not shared, is compiled by default at absolute addresses. The need to randomise the application code is only driven by security requirements.

Non-PIE executables could contain enough gadgets to build very small ROP programs which are able to de-randomise the base of the libc library and then mount a return-to-lib(c) attack. Roglia et al. [34] showed two different attack strategies against non-PIE programs. The authors also concluded that "[...] ASLR is really effective only when used in combination with position-independent executables (PIE)." It is widely accepted that when all sections as well as key data structures are randomised [49], ASLR provides its maximum protection.

6.3 Offset2lib: The Linux ASLR weakness

This section describes a weakness in the design of the ASLR on GNU/Linux. It is specific to GNU/Linux and does not affect Windows or Mac OS. Furthermore, it is not a programming error on the code implementing the ASLR but a weakness in the design. Fortunately, it can be easily fixed on 64-bit systems, as seen in section 6.6.

The problem appears when an application is PIE-compiled. The executable image is handled as if it were a shared library, that is, it is loaded at a random location in memory. The GNU/Linux algorithm for loading ASLR objects works as follows:

1. The first library is loaded at a random position.
2. The next ASLR object is located right below (lower addresses) the previous object.

All ASLR libraries are located ‘side by side’ in a single random place. In the case of PIE applications, the application is also placed in this single-random location. Therefore, an info leak of an address belonging to the application is enough to de-randomise most of the memory map of the application. Note that it is not necessary to have a leak of a GOT address (after it has been properly initialised) but just the program counter of the process.

```
7f36c6a07000-7f36c6bbc000 r-xp  .../libc-2.15.so
7f36c6bbc000-7f36c6dbb000 ---p  .../libc-2.15.so
7f36c6dbb000-7f36c6dbf000 r--p  .../libc-2.15.so
7f36c6dbf000-7f36c6dc1000 rw-p  .../libc-2.15.so
7f36c6dc1000-7f36c6dc6000 rw-p
7f36c6dc6000-7f36c6de8000 r-xp  .../ld-2.15.so
7f36c6fd0000-7f36c6fd3000 rw-p
7f36c6fe5000-7f36c6fe8000 rw-p
7f36c6fe8000-7f36c6fe9000 r--p  .../ld-2.15.so
7f36c6fe9000-7f36c6feb000 rw-p  .../ld-2.15.so
7f36c6feb000-7f36c6fed000 r-xp  /tmp/app-PIE
7f36c71ec000-7f36c71ed000 r--p  /tmp/app-PIE
7f36c71ed000-7f36c71ee000 rw-p  /tmp/app-PIE
7fffe4018000-7fffe4039000 rw-p  [stack]
7fffe41b7000-7fffe41b8000 r-xp  [vdso]
```

0x5e4000

Listing 6.1: Memory map of an application compiled with PIE.

The weakness that we exploit is that the distances between the `app-PIE` application and libraries are always the same in a concrete system. In order words, the distance in bytes from where the application was loaded and where the libraries are mapped is an invariant value in all executions. We named this invariant distance `offset2lib`. It is possible to calculate off-line the offset to each library. For instance, as shown in listing 6.1, from the application text base to the libc text base the `offset2lib` is:

`0x7f36c6feb000-0x7f36c6a07000 =` **`0x5e4000`**

The `offset2lib` is a constant value which may be slightly different on each system. The value depends on the following:

- **The set of libraries used by the application**: depending on the libraries used by the application, the distance between the application base text and the target library could be higher or lower. This information is contained in the executable and can be calculated off-line.

Consequently, the number of libraries is known and is the same for all systems.

- **The version of the each library:** when a new library version is released it is an incremental modification of the previous one, and typically it contains new features or security fixes. These modifications could affect the resulting library size. Typically, different systems use different versions of the libraries, which in turn will affect the distance from where the application is loaded and where the libraries are mapped. As a result, the specific version of a concrete library is the same for all concrete systems.

Note that since the application is mapped at the beginning, its size does not change the `offset2lib` value.

The `offset2lib` between the application and the libraries in non-PIE processes is a random value; in fact, the position of the application is fixed and known at compile time, and what would be needed by an attacker is an address which points to an object or function to any library.

This weakness is especially dangerous because a leak of any address belonging to the application can be immediately used to defeat the ASLR. As detailed in section 6.4, our attacker only needs to make a very simple subtraction from the leaked address.

This weakness could be exploited in both 32- and 64-bit systems, but it is particularly interesting for the latter, due to the high entropy introduced by the ASLR on 64-bit systems, which makes it quite hard to use brute force attacks in practice. A good implementation of the ASLR on 64-bit systems may easily have more than 19 bits of entropy – a level of entropy which will discourage most remote attackers.

Note that *offset2lib* can be exploited by attacks that rely on vulnerability in the application code, not in the libraries or the operating system. As cited by Steve McConnell [50], "A pair of studies performed [in 1973 and 1984] found that, of total errors reported, roughly 95% are caused by programmers, 2% by systems software (the compiler and the operating system), 2% by some other software, and 1% by the hardware. Systems software and development tools are used by many more people today than they were in the 1970s and 1980s, and so my best guess is that, today, an even higher percentage of errors are the programmers' fault." Application code is prone to containing programming errors, and therefore info leaks, which makes the presented vulnerability more dangerous.

6.4 Building the Attack

This section details the steps involved in building a successful attack to bypass the ASLR x86_64 GNU/Linux by exploiting the weakness presented in section 6.3.

Note that the following is only a demonstration example which takes advantage of the ASLR weakness. Section 6.4.4 briefly describes other alternatives used to build more ways of exploiting weaknesses.

Our attack against address-space layout randomisation successfully defeats the ASLR of a PIE-compiled application in just a few attempts, locally and remotely, while Shacham et al.'s attack [14] requires up to 2^n attempts, making the attack unfeasible for 64-bit architectures, or Roglia et al.'s attack [34], which fails in the presence of PIE.

6.4.1 The vulnerable server

To demonstrate the feasibility of bypassing the ASLR, by exploiting our finding, we built a vulnerable target server which was executed in Ubuntu 12.04.3 LTS Linux distribution, equipped with an x86_64 Intel Core i3-370M CPU, clocked at 2.4 GHz and had 3072 MB of RAM.

We introduced a standard stack buffer overflow error, similar to those recently found in the Nginx HTTP Server [51], Ultra Mini HTTPD [52] and PostgreSQL [53, 54], into the target server. The server was implemented as a standard forking server, where each client request is attended by a dedicated child process. This architecture is widely used, due to its simplicity in handling multiple concurrent clients, as well as its stability, security and scalability.

The vulnerable function introduced in the server is shown in listing 6.2. The overflow occurs when a buffer, `str`, larger than `48` bytes, is passed to the `vuln_func()`. It is naively copied into the local vector, `buff`, which the overflows. Also, we consider that the vulnerable function is invoked with the same data sent to the server by clients – attackers in our case. That is, we assume that there is no intermediate cooking or modification of the attacker data.

The server has been compiled and executed with the maximum possible ASLR support from both the compiler and the operating system. Table 6.1 shows information about compilation flags as well as operating system configuration and other protection mechanisms under which our server will be executed.

```
void vuln_func(char *str, int lstr){
   char buff[48];
   int i = 0;
   ...
   for (i = 0; i < lstr; i++) {
      if (str[i] != '\n')
         buff[lbuff++] = str[i];
   ...
}
```

Listing 6.2: Server vulnerable function.

Parameter	**Comment**	**Configuration**
App. relocatable	Yes	`-fpie -pie`
Lib. relocatable	Yes	`-Fpic`
ASLR config.	Enabled	`randomise_va_space = 2`
SSP	Enabled	`-fstack-protector-all`
Arch.	64-bits	x86_64 GNU/Linux
NX	Enabled	PAE or x64
RELRO	Full	`-wl,-z,-relro,-z,now`
FORTIFY	Yes	`-D_FORTIFY_SOURCE=2`
Optimisation	Yes	`-O2`

Table 6.1: Security server options.

Although bypassing the SSP technique, FORTIFY or RELocation Read-Only (RELRO) are not our primary goals, since they can be bypassed without extra complexity, but in the description of this example we decided to enabled them in order to show a more realistic PoC.

As highlighted in listing 6.1, we added extra security flags to the server. Concretely we added the `-fstack-protectorall` GCC flag, which protects not only functions with buffers larger than 8 bytes, but also every function in the application or the GCC flag `-wl,-z,-relro,-z,now`, which in turn removes the possibility of defeating the ASLR by overwriting GOT entries [34].

6.4.2 Steps to building the attack

We structured the attack in five steps. Briefly, our attack starts by analysing off-line the target application and its execution framework. Any missing information (due to ASLR) is obtained via brute force, thanks to the forking server architecture of the target. Once we have the full address of the application, the base address of the application is calculated. The last step involves acquiring the memory map of all the libraries.

With the obtained information it is now easy to arm an ROP program to get a remote shell. The complete on-line attack may take no more than 1 second.

Attack step 1: extract static information

Before going for the address of the target application, it is mandatory to analyse the information that can be obtained off-line, as the result of the analysis will guide and focus the way the attack is carried out. In some cases, it may be possible to obtain some bits of the address off-line, which could then be used to check that the target leaked address is correct (what is expected) or to avoid leaking unnecessary parts of the address (which are already known).

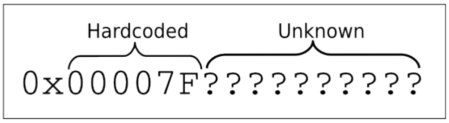


FIGURE 6.1: Saved IP: Hardcoded high bits.

Since in this attack we exploit a stack buffer overflow, we decided to leak the saved IP (Instruction Pointer) of the function, called `vuln_func()`, which is in the stack as the return address of the current stack frame. At first glance, this address might seem to be unknown (fully random), but it is possible to set some high and low bits of the saved IP by just knowing the way the processes are loaded in memory. Concretely, we know that GNU/Linux, as shown in figure 6.1, sets the highest 24 bits to a known value: `0x00007F`.

Additionally, the address we are leaking (saved IP) is used to resume the execution on the instruction following the subroutine call. Hence, by disassembling the executable where the call was made, we can obtain the low bits of the next address to be executed which is located right after the assembler `call vuln_func` instruction. As listing 6.3 shows, these bits correspond with `0x12DF`.

Since the executable has to be aligned to `PAGE_SIZE`, which in the current x86_64 GNU/Linux is 4096 bytes, then the 12 lower bits will not change when the executable is randomly loaded into memory. By doing a simple bit mask operation, we found that the 12 lowest bits are `0x2DF`. The resulting saved IP of setting both highest and lowest known bits is shown in Figure 6.2.

```
0000000000001063 <attend_client>:
1063:   55                           push  %rbp
1064:   48 89 e5                     mov   %rsp,%rbp
1067:   48 81 ec 60 04 00 00         sub   $0x460,%rsp
106e:   64 48 8b 04 25 28 00         mov   %fs:0x28,%rax
1075:   00 00
1077:   48 89 45 f8                  mov   %rax,-0x8(%rbp)
107b:   31 c0                        xor   %eax,%eax
.....   .....  [CALLER_PAGE_OFFSET]  .....
12d7:   48 89 c7                     mov   %rax,%rdi
12da:   e8 1c fc ff ff               callq  efb <vuln_func>
12df:   48 8d 85 c0 fb ff ff         lea   -0x440(%rbp),%rax
12e6:   48 89 c7                     mov   %rax,%rdi
.....   .....         [From the ELF] .....
```

Listing 6.3: Assembly dump of vulnerable server.

Attack step 2: brute forcing the return address

The next step consists of obtaining the remaining 28 random bits of the saved IP address. To obtain these bits we made a fast brute force byte-for-byte attack [20].

The second lowest byte is a 'special byte', because the lower four bits are already known. So, to brute force this byte we fixed these four bits and changed only the four highest ones per attempt. In the worst case it would take only $2^4 = 16$ attempts (0x2, 0x12, 0x22, 0x32 ... 0xF2).

The remaining three bytes were guessed by launching a standard byte-for-byte attack. Note that in order to guess 28 bits we needed to perform $\frac{2^4+3*2^8}{2} = 392$ attempts on average.

The vulnerable function in our server has a buffer size of 48 bytes, for clarity. Assuming a more realistic scenario with a size buffer of 512 or even 1024 bytes, the length of the client request sent to the server will be around 196 or 392 Kbytes, respectively, which is a fairly small number of bytes sent over the net. The temporal cost of the attack is determined by server bandwidth.

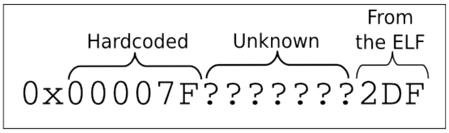


Figure 6.2: Saved IP: Low bits from ELF.

After executing the byte-for-byte attack we particularly obtained the `0x36C6FEC` value. On this point we already knew the saved IP value. Figure 6.3 shows the leaked application address that we obtained.

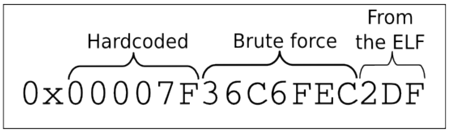


FIGURE 6.3: Saved IP: Brute force bits.

Note that the server was compiled with the stack smashing protector enabled, which forced us to set correctly the stack canary value on every overflow. The value of the canary must be obtained before proceeding to brute forcing the saved IP address.

For clarity, we omitted attacking the SSP, but this protection mechanism can be bypassed by following the same strategy as used to brute force attack the 3 bytes of the saved IP address. Since the first byte is set to zero on Ubuntu, on average it will take $\frac{7*2^8}{2} = 896$ trials to obtain the canary value. Adapting the attack to bypass the SSP, and assuming a buffer of 1024 bytes, the size sent to the server would be approximately 896 Kbytes.

In our attack, the whole time taken to bypass both SSP and ASLR protection is still around 1 second, because the average number of bytes sent to the server is ≈ 60 Kbytes ($896 + 392 = 1288$ attempts $* 48$ bytes). The attack will succeed at around 1 second in systems that are able to handle enough concurrent requests and where there is a bandwidth of greater than 60 Kbytes/s.

Attack step 3: calculating the base application address

In this step we use the leaked address in the previous step to calculate the executable base address. The formula to obtain the base address is:

```
App_base = (savedIP & 0xFFF)-(CALLER_PAGE_OFFSET << 12)
```

where the `savedIP` is the return address value obtained in step 2. The value of the CALLER_PAGE_OFFSET is the number of pages between the executable base and the return address (the address right after the `callq` which invoked `vuln_func`).

In our example, shown in listing 6.3, the next instruction to the `call` is at offset `0x12DF`, which means that the next instruction `lea` is not on the first page but on the second one. Therefore, the value of `CALLER_PAGE_OFFSET` is `1` (the second page). As a result we obtain the base address of the PIE-compiled application, which is:

```
**0x7F36C6fEB000** = (0x7f36C6FEC2DF & 0xFFF) - (1 << 12)
```

Attack step 4: calculating library offsets

The offset value from the base executable application to each mapped library (`offset2lib`) depends on the size and the number of libraries in between. In addition, some applications and libraries may request mmapped memory, before loading the libraries. For instance, as showed in listing 6.1, from the base base address of the application where the dynamic linker is loaded there are two memory mapped areas:

```
[0x7f36c6fd0000 - 0x7f36c6fd3000]
[0x7f36c6fe5000 - 0x7f36c6fe8000]
```

The distance remains unchanged between different executions of the application. Table 6.2 shows some of the offsets from the base of the application to different libraries.

Library	Version	Distance in bytes
Dynamic linker	2.15	`0x225000`
Libc	2.15	`0x5e4000`

Table 6.2: Offsets from executable to libraries

These offsets are different, depending on the system, but measurements show that the values of `offset2lib` are quite similar among this group. Table 6.3 lists the values for different Libc versions in 64-bit systems on some Linux distributions.

Distribution	Libc version	Distance in bytes
CentOS 6.5	2.12	`0x5b6000`
Debian 7.1	2.13	`0x5ac000`
Ubuntu 12.04 LTS	2.15	`0x5e4000`
Ubuntu 12.10	2.15	`0x5e4000`
Ubuntu 13.10	2.17	`0x5ed000`
openSUSE 13.1	2.18	`0x5d1000`

Table 6.3: Offset2lib value on different systems.

Attack step 5: getting app. process mapping

The base address of any library can be calculated by just subtracting the `offset2lib` of the given library from the base of the executable. For instance, to calculate the Libc base address in our example we used the base address of the application obtained in step 3 and the `offset2lib` for the Libc obtained in step 4. The Libc base address for the Ubuntu 12.04 LTS is:

```
0x7F36C6A07000 = 0x7F36C6fEB000 - 0x5E4000
```

as can be verified in listing 6.1. At this point, the ASLR is defeated, and we can repeat the operation to obtain any mapped library of the process.

6.4.3 Exploiting the server target

Although the goal of this paper is to bypass the ASLR, for completeness we briefly describe how we used information to take a remote shell from the vulnerable server.

We obtained the canary value and the base address of the Libc library in previous steps, which allowed us to use the Libc code to build ROP gadgets. In our experiments we found enough gadgets in the Libc 2.15 to build an ROP sequence to execute commands. We built a small script that automates defeating the ASLR and adjusting ROP. The script is able to acquire a remote shell in less than 1 second in all cases.

6.4.4 Other attack vectors

The attack presented in this section is only a demonstration example of the ASLR weakness presented in this paper. We believe that the ways to take advantage of this weakness are only limited by attackers' creativity.

In our attack we performed a small brute force attack against a part of the saved IP address, but other approaches may be available, depending on the manifestation of the vulnerability. For instance, it could use the `printf()`, the `send()` or `write()` library calls to leak an application address by redirecting a pointer.

Some of these functions, such as `send()`, are very robust and gently with incorrect parameters, which can be used to leak application addresses without crashing the system. According to the manual page the process will not crash upon attempts to read from unmapped process areas.

These approaches open up the possibility of leaking application addresses, without crashing the server. Therefore, exploitation is not restricted to forking servers or the like.

6.5 Countermeasures discussion

Obviously, prevention is the best antidote, but as experience shows, it is impossible to write any code that will be free of errors.

Fortunately, the combination of multiple protection techniques has an multiplicative effect. For example, the leaking of an application address may be used to bypass the ASLR and build the correct ROP sequence, but the SSP may prevent it from redirecting the execution flow. In other cases, the control flow can be redirected but the ASLR renders it useless.

The effectiveness of both ASLR and SSP techniques depends on keeping secret some critical information. In the former case, the information is the memory map of the target, and in the latter case it is the value of the canary guard. In both cases, the more entropy, the harder it becomes to guess them.

The entropy concept is quite generic. Typically, it is only associated with the range of values associated with the secret, and it is measured as the number of random bits of the address or the canary. For example, PaX implements a stronger variant of ASLR which does (among other things) exactly this point. On 64-bit x86 machines PaX's ASLR implementation operates with 35 bits of entropy compared to 28 bits on the default Linux implementation.

Unfortunately, some attacks are not blocked by increasing this kind of entropy. Vulnerabilities that are prone to byte-for-byte attacks are only linearly (and not exponentially) improved when more bits of entropy are added. In other words, byte-for-byte attacks are very effective, regardless of the number of random bits to discover.

Nonetheless, there are other *dimensions* of entropy that must also be considered. A recent example of another form of entropy was proposed by Hector et al. [24]. The new technique, called *renew-SSP*, is a variant of the classic SSP technique in which the value of the secret canary is renewed dynamically at key places in the program. This way, the secret is refreshed more often. Rather than refreshing/renewing the secret once per process, it can be refreshed even once per loop. The technique is not intrusive, and it can be applied by just pre-loading a shared library. The overhead, in this instance, is almost negligible.

Beyond the technical aspects, a critical issue to take into account when using new protection techniques is backward compatibility. There is a large amount of code which cannot be upgraded easily, because it is no longer available or maintained. In addition, techniques which introduce a lot of changes in the development process are hardly adopted. Fortunately, the renew-SSP technique is transparent and easy to apply.

The vulnerability used in this paper, to illustrate the weakness of the ASLR, is not exploitable when the renew-SSP technique is used. The same executable image was used both on a standard system (which was defeated in 1 second) and by pre-loading the renew-SSP shared library (would be defeated in several centuries).

Re-randomising the ASLR dynamically at run-time seems to be a rather difficult task. Once the program starts running, living objects (the addresses of structures and functions) refer to current mapping, and the more references/pointers created, the more costly it will be to relocate all of those living objects.

Fortunately, there is room for improvement which is both simple to implement and transparent to existing code. The next section outlines a new implementation of ASLR with more entropy.

6.6 New Full-ASLR design

Current implementations of ASLR operate on a per-group basis. There is a group for *mmapped* areas, another group for the stack, one for the heap, the executable image, etc. All areas in the same group are located side-by-side, starting at a random address. We have designed a new Full-ASLR which is not vulnerable to the attack presented in this paper.

For compatibility reasons it is advisable to implement the new design as a new randomisation mode, configurable through `/proc/sys/kernel/-randomise_va_space`, as the option number 3. Eventually, the proposed implementation will replace the current one (number 2).

0 - No randomisation. Everything is static.

1 - Conservative randomisation. Shared libraries, stack, `mmap()`, VDSO and heap are randomised.

2 - Full randomisation. In addition to elements listed in the previous point, memory managed through `brk()` is also randomised.

3 - Full randomisation, where all or selected elements can be individually randomised. The proposed new ASLR, named 'ASLR-NG', is described in Chapter 7.

6.7 Conclusion

We have presented a new weakness in the current implementation of the ASLR on 64-bit GNU/Linux systems which affects PIE-compiled executables. Applications compiled with PIE are considered to be more robust to attacks, since, for instance, it is not possible to use return-2-* strategies.

We show that it is possible to de-randomise the ASLR, and so defeat it, if an attacker can obtain an application program address. Previous attacks required leaking a pointer belonging to a library. Since the application code is more prone to containing programming bugs, our finding opens up the possibility of exploiting a wider range of errors.

The weakness is illustrated with a detailed proof of concept attack against a vulnerable server (which contains a classic stack buffer overflow), compiled with all security options enabled and with the maximum level of protection. Concretely, we implemented an attack which bypassed the three most widely used and effective protection techniques, namely No-eXecutable bit (NX), address space layout randomisation (ASLR) and stack-smashing protector (SSP). Our attack bypassed the ASLR on a 64-bit GNU/Linux and obtained a remote shell in less than second.

A review of the existing countermeasures that may mitigate the exploitation of this weakness was presented. A holistic defence was considered not only in relation to the ASLR, but also to how other techniques can be used to avoid an attack.

Finally, we proposed an alternative design for the Linux ASLR which removes the disclosed weakness, named 'ASLR-NG'. In this ASLR-NG design, all sections (anonymous maps, libraries, executable images, stacks, etc.) are randomly placed, which adds a new level of entropy: *the distance between maps is also random.* This new ASLR design removes the weakness and thwarts our attack. The new design will be transparent to the applications (no need to recompile the code), and only the Linux kernel shall be modified.

As a final conclusion to this work, despite the fact that there have been great advances in many mitigation techniques, there are still programming bugs (such as the one shown as PoC) that can be successfully exploited. Therefore, it is essential to continue developing new techniques or improving existing solutions.

Chapter 7

ASLR-NG: Address Space Layout Randomisation Next Generation

A taxonomy of all of the elements and constraints that determine the ASLR operation is presented in this chapter. Based on this complete view of the problem a new ASLR design is proposed, called ***'ASLR-NG'****, which outperforms all current ASLR implementations in all aspects.*

Contents

7.1 Introduction

Address Space Layout Randomisation (ASLR) is a well-known, mature and widely used protection techniquewhich randomises the memory address of processes in an attempt to deter forms of exploitation which rely on knowing the exact location of the process objects. Rather than increasing security by removing vulnerabilities from the system, as source code analysis tools [40] tend to do, ASLR is a prophylactic technique which tries to make it more difficult to exploit existing vulnerabilities [55].

The security offered by ASLR is based on several factors [14], including how predictable the random memory layout of a program is, how tolerant an exploitation technique is to variations in memory layout and how many attempts an attacker can make practically.

ASLR is a wide spectrum protection technique, in the sense that rather than addressing a special type of vulnerability, as the renewSSP [24] does, it jeopardises the programming code [23] of the attackers independently of the vector [56] used to inject code or redirect the control flow. Similarly to other mitigation techniques, the ASLR mitigates code execution attacks by crashing the application, and so the attack is degenerated into a denial of service.

The ASLR is an abstract idea which has multiple implementations [57–60], though there are important differences in performance and security coverage between them. We therefore need to make a clear distinction between the core concept of ASLR, which is typically described as something which"*introduces randomness in the address space layout of user space processes*" [4], and the exact features of each implementation.

Although the ASLR is more than 14 years old [31], there is still a lot of work and innovations to be done, both on the design and the implementation. Google has added ASLR to Android 4.0, and PIE support on 4.1. Another area of active work is in the implementation of the KASLR (Kernel ASLR), which loads kernel code and drivers or modules in random positions [61].

The topic of this paper involves a new ASLR design for user processes. The major contributions of this paper are as follows:

- Four different weaknesses are identified in the Linux and PaX ASLR designs.
- ASLRA: a tool used to measure and analyse the quality of ASLR entropy.

	Feature	Description
When	Per-boot	Every time the system is booted.
	Per-exec	Every time a new image is executed.
	Per-fork	Every time a new process is spawned.
	Per-object	Every time a new object is created.
What	Stack	Stack of the main process.
	LD	Dynamic linker/loader.
	Executable	Loadable segments (text, data, bss, ...).
	Heap	Old-fashioned dynamic memory of the process: `brk()`.
	vDSO/VVAR	Objects exported by the kernel to the user space.
	Mmaps/libs	Objects allocated calling `mmap()`.
How	Partial VM	A sub-range of the VM space is used to map the object.
	Full VM	The full VM space is used to map the object.
	Isolated-object	The object is randomised independently from any other.
	Sub-page	Page offset bits are randomised.
	Bit-slicing	Different slices of the address are randomised at different times.
	Direction	Topdown/downtop search side used on a first-fit allocation strategy.
	Specific-zone	A base address and a direction where objects are allocated together.

Table 7.1: Summary of randomisation forms.

- A new ASLR design, named 'ASLR-NG', which outperforms current designs in all aspects.
- An implementation of ASLR-NG in the Linux kernel, showing that it is a realistic replacement of current ASLRs.
- A novel solution for reducing fragmentation, without reducing entropy.

This paper is organised as follows: a brief taxonomy of the ASLR is presented in section 7.2, followed by a critical analysis of the constraints that have historically determined the design of the ASLR and which are the root causes of the weaknesses presented in section 7.4. We then describe in section 7.5 the constraints that must be taken into account when designing a practical ASLR, and the new ASLR-NG is presented in section 7.6 and evaluated in section 7.7. Section 7.8 concludes the paper.

7.2 System model and definitions

Depending on the exact implementation details there may be important differences in the final operation and effectiveness of the ASLR, and so in order to understand and compare these differences between ASLR implementations, it is necessary to have a detailed definition of all memory objects and how they can be randomised.

In what follows, a *memory object* is classed as a block of virtual memory allocated to a process, examples of which include the stack, the executable, a mapped file, an anonymous map and the vDSO. For our purposes, the size and the base address of each object are the most relevant attributes.

Several objects may be allocated together (in consecutive addresses) with respect to a random address base, which will be referred to as the *area* or as the *zone*. For example, in Linux, all objects allocated via the `mmap()` system call are placed side by side in an area (*mmap_area*).

ASLR entropy can be categorised, as shown in Table 7.1, into three main dimensions: 1) When, 2) What and 3) How. The first dimension defines how often randomisation takes place, the second determines which objects are randomised and the last one defines how and how much the objects are randomised.

Regarding the **When** dimension, the more frequent, the better, i.e. per-exec randomisation is preferred to per-boot, and per-fork is better than per-exec, etc. The entropy used to decide where an object is going to be placed has been taken at boot, when a new image is loaded, etc. For example, Linux ASLR randomises all objects per-exec, so once the process has been created, all subsequent objects (mmap requests) are located side by side, and so no new entropy is introduced.

The second entropy dimension is granularity (i.e. **What** is randomised), whereby the more objects that are randomised, the better. Some security researchers consider that if a single object (for example, the executable) is not randomised, the ASLR can then be easily defeated. We will assume that all objects are randomised, but when and how each one is randomised may differ between implementations.

The way an object is randomised is defined by the last dimension: **How**. It is possible to consider two sub-dimensions: 1) how many bits[1] are random and 2) what is the randomness between objects (inter-object). That is, the absolute entropy of the object by itself and the conditional entropy between objects.

Regarding how many bits of the address are randomised, there are two forms: **full-VM** and **partial-VM**. Partial-VM is when memory space is divided in disjointed ranges to generate random numbers for the addresses. In Full-VM complete virtual memory space is used to randomise an object. The requirements required to carry out a full-VM are analysed in the next sections. As far as we know, current ASLR implementations only use partial-VM randomisation.

[1]Historically, entropy has been measured in 'numbers of bits', but it would be more accurate to consider distribution and its parameters.

The implementation of the ASLR relies strongly on the processor virtual memory infrastructure (memory page), which greatly simplifies the randomisation of page addresses. The sub-page randomisation form refers precisely to randomising the bits that belong to page alignment. The other four forms of entropy (isolated-object, bit-slicing, direction and specific-zone) are presented and discussed in detail in the following sections.

7.3 Growable objects: a critical review

In this section we analyse the problems and limitations caused by growable objects and the available techniques to make them more ASLR-compatible.

The Linux and PaX[2] ASLR designs rely on the same core ideas, in that they define four partial-VM areas: 1) stack, 2) libraries/mmaps, 3) executable and 4) heap.

The classic memory layout was designed by considering that some zones or objects are *growable* (main stack, thread stacks and heap). In order to allow them to grow, they have to be placed within the extreme reaches of virtual memory, far away from other objects, otherwise they will not grow or, even worse, silent collision could occur.

Originally, the functionality of growable objects was a smart, simple and efficiently solution for efficient memory usage. However, the use of threaded applications and the possibility of adding dynamically new objects into the memory space forced developers to reconsider the viability of these growable areas. Today, growable objects are a source of numerous problems [62, 63], but fortunately a set of advanced programming solutions has been developed which removes the necessity of this type of object.

Growable objects impose strong limitations on ASLR design, and they affect negatively the entropy of all objects. The situation gets even worse when multiple growable objects are used in the application, as actually happens with multiple thread stacks. The approach used in Linux, (see Figure 7.1) involves placing each object as separately as possible from each other, which forces to fix high bits of the addresses, thus degrading effective entropy. Since the extremes of the virtual memory are already occupied, libraries and mmaps are placed between the stack and the heap. Note that a small shared library is automatically mapped by the kernel into the address space of all user-space applications (vDSO). Therefore, both static and dynamically linked (PIE or not) programs have a similar memory layout.

[2]FreeBSD, HardenedGentoo and others use the PaX ASLR approximation.

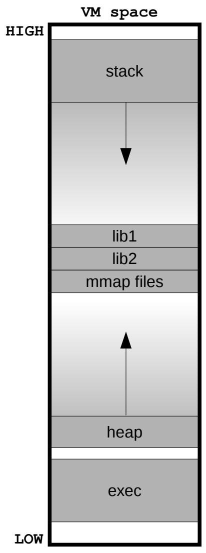


FIGURE 7.1: Classic memory layout.

Originally, PIE-compiled applications were loaded jointly with the libraries, but after the Offset2lib weakness [25] was identified, the PIE executable was moved to lower addresses (Linux 4.1) and its own zone.

7.3.1 Stacks

There are two different kinds of stacks, namely the stack of the main process, and the stack of the threads or clones (since both thread and cloned entities handle stacks in a similar way, in what follows we will refer to them jointly as 'thread stacks'). The main stack is still considered and handled as a growable object.

Initially, thread stacks were 'set up' to be growable. Flags `MAP_GROWSDOWN` and `MAP_GROWSUP` were added to the `mmap()` request, to tell the kernel about expected behaviour. Inevitably, these flags were removed [62] because of security problems and intrinsic logical limitations.

Nowadays, thread stacks are treated as regular (non-growable) objects, reserved with the maximum estimated size when the threads are created. By default, the thread stack size is set to 8MB (in Debian and Ubuntu), but it can be changed by using the `RLIMIT_STACK` resource with the `setrlimit()` system call. Note that the `RLIMIT_STACK` value is used as the default thread stack size rather than an upper limit.

A summary of the facilities provided by the compiler, to deal with growable stack issues, is presented below:

- One or more protected pages (page guards) are placed at the end of the thread stack. If the stack overflows, then the process receives a SIGSEGV signal. This guard is further enforced by the GCC flag `-fstack-check`, which emits extra code to access sequentially all the pages of the stack, thus preventing overflowing by jumping over the page guard.
- The split stack feature (GCC flag `-fsplit-stack`) generates code to automatically continue the stack in another object (created via `mmap()`), before it overflows. As a result, the process has discontinuous stacks which will only run out of memory if the program is unable to allocate more memory. This is an interesting feature for threaded programs, as it is no longer necessary to calculate the maximum stack size for each thread. This is currently only implemented in the i386 and x86_64 back-ends running in GNU/Linux.
- It is possible to ask the compiler to print stack usage information for the program, on a per-function basis, using the `-fstack-usage` flag and making an estimation of stack size.

Although these facilities are very helpful when dealing with stacks, in practice most applications are able to work using default stack size (Google Chrome(r), LibreOffice, Firefox, etc.). Only very demanding applications have stack size issues, which are typically handled by slightly increasing the `RLIMIT_STACK` limit value. For example, Oracle(r) advises to set it to 10MB when running its database.

7.3.2 The heap

When the process needs more heap memory, it calls the `brk()` system call to move forward (higher addresses) the heap's end. The `brk()` request may fail if 1) there is not enough free memory contiguous to the existing heap, because the end of the memory has been reached or because another object is already in that address, or 2) the data segment limit has been exceeded, as set by the `RLIMIT_DATA` resource.

The heap is exclusively used by the standard C library to provide the dynamic storage allocation (DSA) service: `malloc()` and `free()`. Although originally DSA algorithms relied exclusively on heap memory, current implementations use multiple non-contiguous objects of memory requested by `mmap()`. In fact, the GNU libc uses mmapped objects when the requested size is larger than 128Kb.

The `brk()` service (i.e. the idea of a memory area close to the executable that can grow at will) has been made obsolete by mmap functionality, marked as LEGACY in SUSv2 and removed in POSIX.1-2001.

7.4 ASLR design weaknesses

Current ASLR designs are influenced heavily by growable area requirements and by compatibility misconceptions. We have identified the following weaknesses.

7.4.1 Non-full address randomised weakness

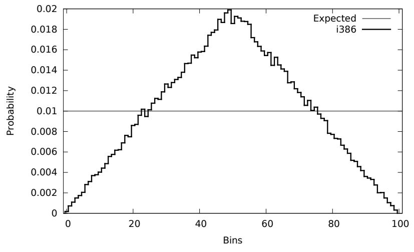
ASLR has been implemented by slightly shaking or moving randomly the base address of objects with respect to the classic layout, where the main stack is at the top, the executable is at the bottom, the heap follows the bss segment and the mmap zone is located in between the heap and the stack. Therefore, entropy that can be applied to each object is limited by the range that they can be moved while preserving the previous sequence. This affects the higher bits of the addresses [14].

Another constraint that reduces entropy is the unnecessary alignment of some objects. Although alignment is mandatory in some cases (huge pages, executable, libraries, etc.), there many are others in which this is not required; for example, many private anonymous maps could have sub-page entropy.

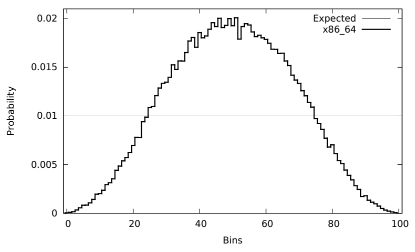
7.4.2 Non-uniform distribution weakness

The distribution of objects along the range should be as uniform as possible. That is, all the addresses should have the same, or very similar, probability of occurrence; otherwise, it would be possible to speed up attacks by focusing on the most frequent (likely) addresses.

Figure 7.2 shows the output of the ASLRA (ASLR analyser) tool for the libraries and mmap objects in PaX. As can be seen, the distributions of these objects are far from uniform, because on i386 it follows a triangular distribution and on x86_64 an Irwin-Hall with $n = 3$. In Linux, the heap is the result of the sum of two random values, but since one of them is much larger than the other, the impact on the distribution is almost negligible. Hence, the final locations shall never be the sum of multiple random values.



(a) i386



(b) x86_64

FIGURE 7.2: Distribution of mmapped objects on PaX.

7.4.3 Correlation weakness

Attacks launched to bypass the ASLR are becoming more and more sophisticated; for instance, instead of attacking an object directly, the attacker can de-randomise the object with low randomisation first, and then use it to de-randomise the target object (the object which contains the required gadgets or data). The idea that an object's memory address leak can be used to exploit another one was first demonstrated by Hector et al. [25] through the offset2lib weakness. In that case, the executable was de-randomised using a byte-for-byte attack [20] and then the libraries zone was calculated, resulting in a very fast bypass of the ASLR.

In Linux and PaX, the heap and the executable are separated from each other by a random value. A leak in the heap area not only compromises the heap, but it also reveals information about the executable, because there is less entropy distance from the executable to the heap (correlation entropy) than the absolute entropy of the executable. Huge pages and the objects in the mmap_area are also correlated. Since huge pages have the largest alignment, they have the lowest entropy, and attackers can build correlation attacks by de-randomising huge pages first and then later the libraries zones. For example, in PaX i386, instead of attacking the libraries directly (16 bits), attackers can de-randomise huge pages (6 bits) and later de-randomise the libraries zone from the huge page zones (10 bits). This two-step attack takes $1088 = 2^{10}+2^{6} \approx 2^{10}$ attempts, instead of $65536 = 2^{16}$ involved in the direct attack on the library.

All mmapped objects are located together in the mmap area, which results in total correlation between all of them, but from a security point of view, this is a weak design. MILS (multiple independent levels of security/safety) criteria state that objects of different criticality levels must be isolated. Google Chrome, for instance, is aware of this security issue and has addressed it by implementing some form of user-land ASLR to map JIT (just-in-time compiled code) objects in its own zone, which is separated from the default mmap_area. Note that JIT objects are an appealing target for attackers [64].

7.4.4 Memory layout inheritance weakness

All child processes inherit/share the memory layout of the parent. This is the expected behaviour, since children are an exact copy of the parent's memory layout. Unfortunately, though, from a security point of view, this is not a desirable behaviour, because although new objects belong only to the child process, their addresses can be guessed easily by parents and siblings.

This problem is especially dangerous on networking servers which use the forking model. In Android, for instance, the situation is even worse, because all of the applications are children of Zygote, and although the siblings might not call the same mapping sequence, a malicious sibling can predict future mmaps of any other. Therefore, the leakage of an object in the library or mmap area exposes all objects in these areas (correlation weakness) and also allows one to predict where future mmaps will be placed – even between siblings (inheritance weakness).

7.5 ASLR constraints and considerations

The straightforward solution to solving most of the previous weaknesses is to randomise each object independently over the full VM range. Although this idea is quite intuitive [59], multiple practical issues must be addressed properly, in order to achieve a realistic ASLR design. ASLR-NG has been designed by taking into account the following issues:

Fragmentation: although, from the point of view of security, having objects spread all over the full VM space is the best choice, in some cases it introduces prohibitive fragmentation, which is especially severe in 32-bit systems. Applications that request large objects or make a lot of requests may fail randomly, so it is mandatory to have a mechanism to address this fragmentation.

Page table footprint: a very important aspect that is underestimated is the size of the process page table, because the more the objects are spread, the bigger the page table. This is particularly important in systems with low memory or with a high number of processes. Since each application could have a different level of security, the ASLR design should allow for tuning the page table size versus object spreading.

Growable areas: unfortunately, most applications still use growable areas in some objects. In order to be compatible with these applications, an ASLR must guarantee some form of compatible behaviour.

Homogeneous entropy: all objects should have the same amount of entropy, in particular objects of the same type (for example, stacks); otherwise, attackers will focus on the weakest link. Unfortunately, none of the current designs meets this requirement.

Uniformly distributed: all objects should be uniformly distributed; otherwise, attackers can design more effective attacks by focusing on the most frequent addresses.

ASLR compatibility: the ASLR design should be backward-compatible with existing applications. That is, if there is a trade off between security and compatibility, then the design should allow for tuning the application framework to meet application's needs.

7.6 ASLR-NG

This section describes the proposed ASLR-NG, which addresses all the weaknesses identified in section 7.4 as well as all of the constraints and considerations presented in section 7.5.

In order to design the ASLR-NG, growable objects (main stack and heap) must be bounded when they are created. If they need to grow, then discontiguous solutions should be used or a larger bound (limit) must be set. As presented in section 7.3, nowadays both the stack and the heap can be handled as non-growable objects, which enables us to propose ASLR-NG. The default value of the stack limit is 8MB, which fulfils the requirements of most applications. More demanding applications can increase this limit. ASLR-NG uses this value to reserve both stack memory and the initial heap.

7.6.1 Allocating object strategy

Two methods are available to allocate an object in ASLR-NG.

1) Isolated: the object is independently randomised using the full virtual memory space of the process. Unlike current implementations, ASLR-NG can use the full VM range to allocate an object, and as a result there no order to the objects and it prevents any kind of correlation attack.

2) Specific-zone: objects of the same class are mapped together and isolated from others. A specific-zone is defined by a base address and a direction flag, both of which are initialised when the specific-zone is created (see function `new_zone()` in Listing 7.1). The base address is a random value taken from the full VM space, and new objects are placed by following the direction flag (toward higher or lower addresses) with respect to the base address.

The main benefit of using specific-zones is that it reduces both fragmentation and page table footprint, which makes the ASLR practical and realistic. Furthermore, specific-zones can be created according to MILS criteria, in that objects of the same criticality level may be grouped together. Criticality depends, among other factors, on the permissions and the kind of data stored on the object. Following this rule, ASLR-NG defines five specific-zones (depending on the configuration, see profile modes below):

Huge pages: placing all huge page objects in their own specific-zone removes the correlation weakness between huge pages and mmapped objects. This is a specially dangerous form of correlation weakness as described in subsection 7.7.4.

Thread stacks: following the same criteria as the main stack, the thread stacks are isolated from the rest of objects on their own specific-zone.

Read-write-exec objects: although these types of object are seldom used, for example in JIT mapping, they are very sensitive; in fact, Google implements custom randomisation in their Chromium browser for these objects as part of its sand-boxing framework.

Executable objects: map requests with executable permission are grouped in a specific-zone. This zone is mainly used to group library codes.

Default zone: any other objects that do not match previous categories are allocated to this specific-zone.

In addition, applications can create custom specific-zones to isolate sensitive data. For example, the credentials or certificates of a web server can be isolated from the rest of the regular data. This mechanism can prevent a Heartbleed [65] attack by moving sensitive data (certificates) away from the vulnerable buffer.

7.6.2 Addressing fragmentation

When virtual memory size is small, fragmentation is an issue, because the more objects that are independently randomised, the more fragmented the memory. In dynamic memory, the fragmentation problem is defined as [66] “the inability to reuse memory that is free.”

There is no simple way to measure fragmentation, but the worst case depend on: 1) the number of objects already allocated, 2) their size, 3) the relative position of each one and 4) the size of the new request. If all objects, n, are independently randomised, the worst case occurs when the allocated objects are of one page size and they are evenly distributed along the whole memory space. In this case, the maximum guaranteed size is approximated by:

$$\text{new\_obj\_size} \lesssim \frac{\text{VM\_SIZE}}{n+1}$$

On x86_64 fragmentation is not a issue because of the very large number of mapped objects needed to cause an error. For example, a 1GB memory request will not fail until $2^{17} = 131072$ objects have been mapped.

On the other hand, fragmentation is a real problem in 32-bit systems. For example, a memory request of 25MB is not guaranteed after just 122 requests (of page size), while a request for 256MB may fail after mapping just 12 objects, including the stack, vDSO, executable, heap, each library, etc. Therefore, it is not practical to randomise each object independently in 32-bit systems, without addressing the fragmentation issue.

ASLR-NG addresses this issue by reserving a range of virtual space, the amount of which is specified as a percentage of the available VM size. When a requested object does not fit into the non-reserved space, the allocation algorithm automatically uses the reserved space, without degrading the entropy of these objects and regardless of their size.

Figure 7.3 shows the result of allocating multiple objects in ASLR-NG. Objects 1 and 3 fit into the non-reserved area, and so they are placed there, but for objects 2 and 4, there are no free gaps to hold them on the non-reserved area. In this case, the algorithm performs a top-down, first-fit strategy. Note that objects allocated in the reserved area will ‘inherit’ the entropy of the lowest object in the non-reserved area.

Although reserving a percentage of the VM will reduce the range for available randomisation, ASLR-NG uses a novel strategy to regain lost entropy, whereby the reserved area is randomly placed at the top or the bottom of the virtual memory space.

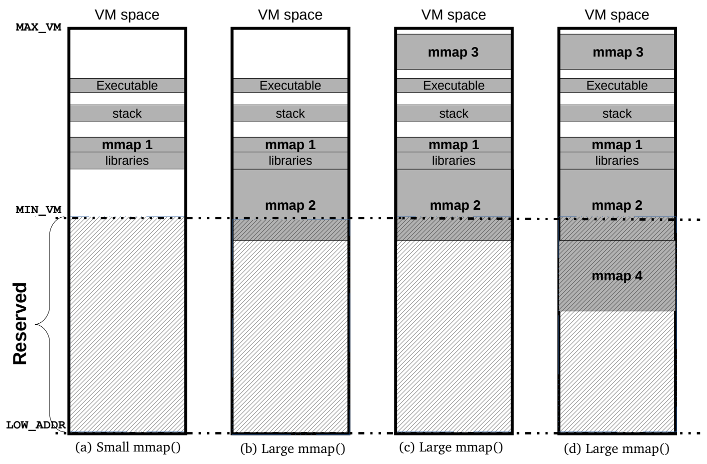


FIGURE 7.3: ASLR-NG: A 50% example of a reserved area.

For example, by reserving 50%, an attacker cannot know on which side (top or bottom) the objects will be located, which forces them to consider the whole VM space. As a result, there is no entropy penalty with this strategy.

Only when the reserved area is larger than 50%, is there a small amount of entropy degradation. The expression which relates the loss of entropy to the percentage of reserved area is:

$$f(x) = \begin{cases} 0, & \text{if } x \leq 50\% \\ -1 - \log_2\left(1 - \frac{x}{100}\right), & \text{otherwise} \end{cases}$$

where x is the percentage of the reserved area, and $f(x)$ gives the number of bits that have to be subtracted from the total VM space entropy.

For example, reserving 50% on an i386, the largest guaranteed object is 1500MB and entropy is not reduced. If 2/3 of the VM space is reserved, then it is possible to allocate an object up to 2GB in size, and at the cost of reducing entropy by only 0.5 bits.

Therefore, the ASLR-NG design has both more entropy and less fragmentation.

7.6.3 Algorithm

When a process is created, the area reserved to avoid fragmentation is defined by setting the variables min_ASLR and max_ASLR. This is the range that will be used to allocate objects (allocation area).

```
new_zone(low, high, zone) {
    zone.base      = randomize_range(low, high);
    zone.direction = randomize_range(low, high) <
                zone.base ? TOPDOWN : DOWNTOP;
}
do_exec(){
    ...
    reserved  =  VM_SIZE * percentage_reserved / 100;
    min_ASLR  = reserved * (rand() % 2);
    max_ASLR  = min_ASLR + VM_SIZE - reserved;
    new_zone(min_ASLR, max_ASLR, mmap_base);
    new_zone(min_ASLR, max_ASLR, huge_pages);
    new_zone(min_ASLR, max_ASLR, thread_stacks);
    ...
}
```

LISTING 7.1: ASLR-NG initialisation pseudo-code.

The direction of a specific-zone is a random bit with a probability of pointing towards the middle of the allocation range inversely proportional to the distance of the base address to the middle – the expression is in Listing 7.1. In other words, if the base address is close to the border of the allocating range, then the direction is more likely to point toward the other side of the range. This way, objects will not accumulate at the borders of the allocation area.

A detailed analysis of the distribution of the objects at the borders of the allocation area is beyond the remit of this paper, but for now we can say that the presented algorithm to determine the direction gives a fair distribution along the whole range, with no accumulation areas (addresses with higher probability), regardless of the number of objects in the zone and the workload mix.

The algorithm employed to allocate an object works by first selecting a value as a hint address, in order to place the object, and then to look for a free gap in which to actually place the object. The algorithm is as follows:

1. Obtain the hint address and the direction:
    - if it is to a specific-zone, then the hint address and the direction are the ones from the specific-zone.
    - if it is an isolated object, then the hint address is a random value from the allocation range [min_ASLR, max_ASLR] and the direction is top-down.

2. Look for a gap large enough to hold the request from the hint address to the limit of the allocation area determined by the direction. If found, then succeed.
3. Look for a gap large enough to hold the request from the hint address to the limit of the allocation area determined by the reverse direction. If found, then succeed.
4. Look for a gap large enough to hold the request from the full VM space, starting from the allocation area and working towards the reserved area. If found, then succeed.
5. Out of memory error.

Even if there is no reserved area, step 4 is necessary to guarantee that the whole virtual memory is covered properly. For example, as illustrated in the Figure 7.4.d, the gaps [ld.so $\leftrightarrow$ mmap_base] and [mmap_base $\leftrightarrow$ vDSO] are not suitable for a large request, but the gap [ld.so $\leftrightarrow$ vDSO] can be used if a global search is done.

7.6.4 Profile modes

The basic ASLR-NG design provides two possibilities for allocating each object: isolated or in a specific-zone. From a security point of view, the more isolated objects, the better, but there are multiple side effects that should be carefully considered and balanced, as described in section 7.5. In order to simplify the configuration of ASLR-NG, we provide four different working modes or profiles. Each mode randomises each object using the isolated or the specific-zone method. The four modes are summarised in Table 7.2, and a representative example of each one is sketched in Figure 7.4. Next is the design rationale for each mode:

Mode 1 - Concentrated: all objects are allocated in a single specific-zone, which results in a compact layout. The number of entropy bits is not degraded but only the correlation entropy between them. In other words, the cost (if brute force were used) to obtain the address of an object is not reduced by using this mode. The goal is to reduce the footprint of the page table.

Mode 2 - Conservative: this mode is equivalent to that used in Linux and PaX. The main stack, the executable and the heap are independently randomised, while the rest (libraries and mmaps) are allocated in the mmap specific-zone. Since the objects are randomised using the full allocation range, ordering is not preserved; for example, the stack may be below the executable.

	Modes			
Feature	**1**	**2**	**3**	**4**
Sub-page in ARGV	✓	✓	✓	✓
Randomise direction	✓	✓	✓	✓
Bit-slicing	✓	✓	✓	✓
Isolate stack, executable and heap		✓	✓	✓
Specific-zone for huge pages		✓	✓	
Randomise specific-zones per child			✓	✓
Sub-page in heap and thread stacks			✓	✓
Specific-zone for thread stacks			✓	
Specific-zone for read-write-exec objects			✓	
Specific-zone for exec objects			✓	
Isolate thread stacks				✓
Isolate LD and vDSO				✓
Isolate all objects				✓

Table 7.2: ASLR-NG mode definition.

Mode 3 - Extended: this is an extension of the conservative mode, with additional randomisation forms: 1) specific-zones for sensitive objects (thread stacks, heap, huge pages, read-write-exec and only executable objects); 2) sub-page randomisation of the heap and thread stacks and 3) per-fork randomisation.

This can be considered a very secure configuration mode which addresses most of the weaknesses and sets a reasonable balance between security and performance. Therefore, this should be the default mode on most systems.

Mode 4 - Paranoid: every object is independently randomised, and no specific-zones are used. As a result, there is no correlation between any objects, which could even prevent future sophisticated attacks. It is intended to be used on processes that are highly exposed, for example networking servers, but should be carefully used when applied globally to all system processes because of additional memory overheads.

7.6.5 Fine grain configuration

Each profile mode is defined by a set of features. The following Table 7.2 lists the ASLR-NG configuration options enabled on each mode.

Sub-page in ARGV: ASLR-NG randomises all the sub-page align bits. Although the arguments/environment are in the stack area, the page align bits of ARGV can be randomised.

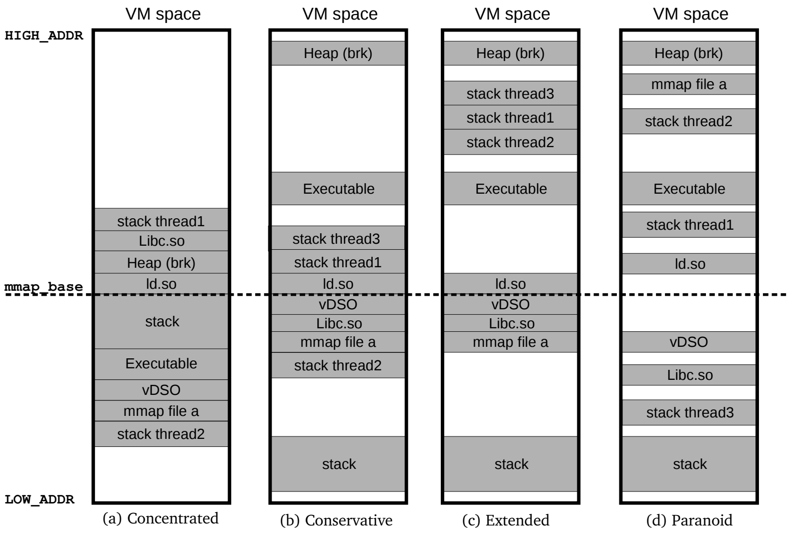


FIGURE 7.4: ASLR-NG: Profile mode examples.

Randomise direction: the direction of a specific-zone is re-randomised for every new allocation. As a result, even libraries that typically are loaded sequentially will have some degree of randomness, which is especially useful in the concentrated profile, because it shuffles objects.

Specific-zone for huge pages: if enabled, ASLR-NG uses a different specific-zone to map huge pages.

Specific-zone for thread stacks: If enabled, thread stacks are allocated in a designated specific-zone.

Inter-Object to Stack, Executable and Heap: each one of these objects is independently randomised, which is the default behaviour for Linux and PaX. It was added to support the concentrated mode by disabling it.

Randomise specific-zones per child: When a new child is spawned, all specific-zones are renewed, which results in a different memory map between the parent and the child, as well as any siblings among them.

Sub-page in heap and thread stacks: applies sub-page randomisation to the thread stacks and the heap. This feature can also be used from user-land on a per-object basis, by calling the `mmap()` with the new flag `MAP_INTRA_PAGE`.

Isolate thread stacks: randomises thread stacks individually. This feature can also be requested by using the `MAP_RND_OBJECT` flag when calling `mmap()`.

Isolate LD and vDSO: by enabling this feature, ASLR-NG loads these objects individually instead of using the classic library/mmap zone.

Bit-slicing: enabling this feature, ASLR-NG generates a random number at boot time which is later used to improve the entropy of some objects when they must be aligned, typically for cache aliasing performance. Instead of setting the sensitive bits to zero, they are set to the random value generated at boot.

We have used the core idea of this novel randomisation form to address a security issue in the Linux kernel 4.1, to increase entropy by 3 bits in the AMD Bulldozer processor family [67].

Isolate all objects: all objects are independently randomised. The leakage of any object cannot be used to de-randomise any other. This feature can be used in very exposed or critical environments where security is paramount.

7.7 Evaluation

This section compares ASLR-NG with Linux and PaX. Firstly, subsection 7.7.1 compares the main randomisation forms to identify the new features introduced by the ASLR-NG. In subsection 7.7.3 the entropy bits for 32 and 64 bits in the x86 architecture are compared, and finally the correlation entropy of the objects is presented.

7.7.1 randomisation forms

Linux and PaX provide very few randomisation forms, and furthermore they do not generalise them either. For example, they do not provide sub-page or inter-object randomisation for thread stacks.

ASLR-NG extends already used forms of entropy to most objects and provides new forms to prevent correlation attacks [25]. It worth mentioning the concept of specific-zones, which is a simple mechanism employed to group together sensitive objects and isolate them from the rest. Table 7.3 summarises the main features of Linux, PaX and ASLR-NG.

Feature and forms	**Linux**	**PaX**	**ASLR-NG**
Inter-object in stack, exec. and heap	✓	✓	✓
Sub-page in main stack	✓	✓	✓
Sub-page in ARGV and heap (brk)		✓	✓
Inter-object in LD and vDSO			✓
Inter-object in thread stacks			✓
Sub-page in thread stacks			✓
Load libraries order randomised			✓
Multiple specific-zone support			✓
Randomise specific-zones per child			✓
Bit-slicing randomisation			✓
Sub-page per mmap request			✓
Inter-object per mmap request			✓
Uniform distribution			✓
Full VM range			✓

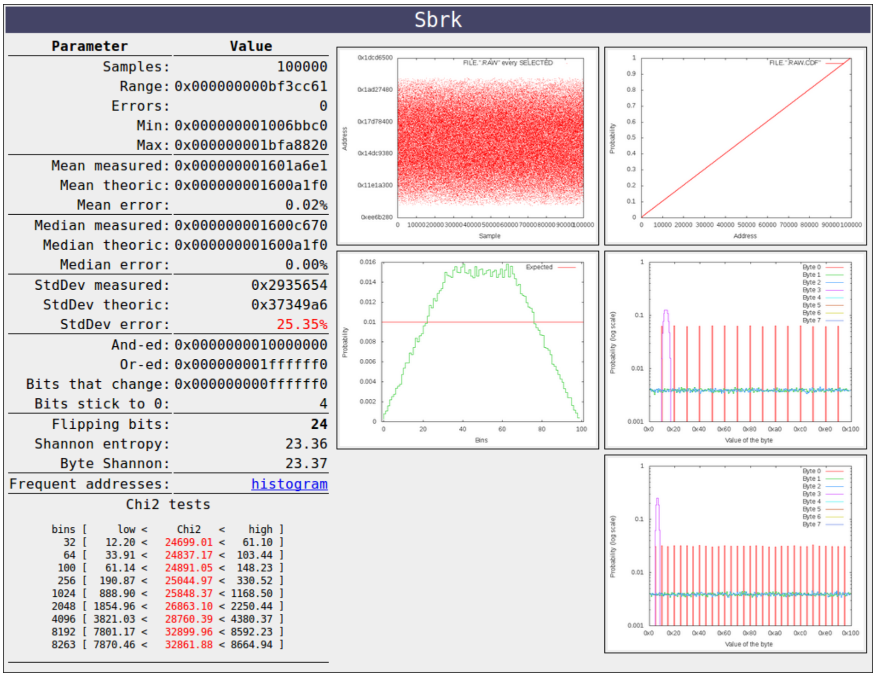
TABLE 7.3: Comparative summary of features.

7.7.2 ASLRA: ASLR Analyser tool

Although it is possible to analyse code and determine any entropy expected from them, there are too many interactions and dependencies between the code that generate random values and code that finally assigns the address to the object. In fact, results from externally observed entropy have been used to detect several defects in both Linux and PaX implementations [67–69].

The PaX team developed a tool called `paxtest` (included in most Linux distributions) to estimate, among other features, the entropy of objects. It uses a custom ad hoc algorithm to guess effective entropy bits. This algorithm has been designed assuming that the underlying distribution is uniform with a power of 2 range. When these conditions do not hold, the result is incorrect. Also, it does not provide basic statistical information about the observed distribution. For example, PaX suffers from non-uniform weaknesses (see section 7.4.2) on most mappings, which are not detected by the paxtest, and entropy is overestimated.

We have developed ASLRA, a test suite, which can be used to measure and analyse the entropy of objects. ASLRA is composed of two utilities: 1) a sampler program, to collect the addresses of objects, and 2) an analyser utility, to perform specific statistical analysis and display the results. Besides the basic statistical parameters (range, mean, median and standard deviation), the analyser calculates three different entropy estimators: flipping bits, individual byte Shannon entropy and standard Shannon entropy.


Sbrk

Parameter	Value
Samples:	100000
Range:	0x000000000bf3cc61
Errors:	0
Min:	0x000000001006bbc0
Max:	0x000000001bfa8820
Mean measured:	0x000000001601a6e1
Mean theoric:	0x000000001600a1f0
Mean error:	0.02%
Median measured:	0x000000001600c670
Median theoric:	0x000000001600a1f0
Median error:	0.00%
StdDev measured:	0x2935654
StdDev theoric:	0x37349a6
StdDev error:	25.35%
And-ed:	0x0000000010000000
Or-ed:	0x000000001ffffff0
Bits that change:	0x000000000ffffff0
Bits stick to 0:	4
Flipping bits:	**24**
Shannon entropy:	23.36
Byte Shannon:	23.37
Frequent addresses:	histogram

Chi2 tests

```
bins [     low <     Chi2   <    high ]
  32 [   12.20 <   24699.01 <   61.10 ]
  64 [   33.91 <   24837.17 <  103.44 ]
 100 [   61.14 <   24891.05 <  148.23 ]
 256 [  190.87 <   25044.97 <  330.52 ]
1024 [  888.90 <   25848.37 < 1168.50 ]
2048 [ 1854.96 <   26863.10 < 2250.44 ]
4096 [ 3821.03 <   28760.39 < 4380.37 ]
8192 [ 7801.17 <   32899.96 < 8592.23 ]
8263 [ 7870.46 <   32861.88 < 8664.94 ]
```

FIGURE 7.5: ASLR analyser: Screenshot of PaX Heap (brk)

For our purposes, the most interesting entropy estimator is standard Shannon entropy, which is calculated by the following expression, where $p(x)$ is the estimated probability of the address x:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x)$$

The resulting entropy value is a good measure of dispersion or 'surprise', but it must be interpreted with caution [70]. In most cases, it is an accurate estimation of the cost of an attack, but only if it is a uniform distribution. It is especially problematic for those distributions with a high kurtosis, because the attacker can focus on a small range of values, thereby building faster attacks. A detailed analysis of attacking strategies versus statistical distributions is beyond the scope of this paper.

	32-bits			64-bits		
Object	**Linux**	**PaX**	**ASLR-NG**	**Linux**	**PaX**	**ASLR-NG**
ARGV	11	27	31.5	22	39	47
Main stack	19	23	27.5	30	35	43
Heap (brk)	13	23.3	27.5	28	35	43
Heap (mmap)	8	15.7	27.5	28	28.5	43
Thread stacks	8	15.7	27.5	28	28.5	43
Sub-page object	-	-	27.5	-	-	43
Regular mmaps	8	15.7	19.5	28	28.5	35
Libraries	8	15.7	19.5	28	28.5	35
vDSO	8	15.7	19.5	21.4	28.5	35
Executable	8	15	19.5	28	27	35
Huge pages	0	5.7	9.5	19	19.5	26

TABLE 7.4: Comparative summary of bits of entropy.

7.7.3 Absolute address entropy

Absolute entropy is the effective entropy of an object when it is considered independently.

Each ASLR has been tested in two different systems:

- **32-bits:** a 32-bit x86 architecture, without PAE. Note that when an i386 application is executed in a x86_64 system, the memory layout is different. Our experiments are executed in a truly 32-bit system, and so the virtual memory space available to any process is 3GB.
- **64-bits:** a 64-bit x86_64 architecture. The virtual memory space available for the user is 2^{47} bytes.

Table 7.4 shows the measured entropy bits obtained in Linux, PaX and ASLR-NG in both 32- and 64-bit systems. All the data presented in this section are the result of running the sampler tool to collect a million samples for each system. ARGV is the page in memory that hold the program arguments.

Linux: In 32-bit systems, Linux provides only 8 random bits for most objects, which is too low a value to be effective and can be considered defeated. In 64 bits, although randomisation is higher for most objects, there are still some objects (vDSO and ARGV) with much lower entropy, which in turn may encourage attackers to use them.

Huge pages are less randomised, due to alignment constraints. In particular, in 32-bit systems, alignment resets those bits that the ASLR randomises, and so huge pages are not randomised at all. Moreover, in 64-bit systems, huge pages have 19 random bits, which gives some protection but still may not deter local or remotely distributed attackers.

PaX/Grsecurity: In 32 bits, PaX provides much more entropy than Linux in all objects. The libraries and mmapped objects have 15.72 bits of entropy, in which case a brute force attack, at 100 trials per second, will need a few minutes to bypass the PaX ASLR. The lowest randomised object (but huge pages) is the executable. Surprisingly, its entropy is smaller than in 64-bit Linux. The additional entropy bits of the ARGV, main stack and heap are due to sub-page randomisation. The decimal values of the mmapped objects are caused by non-uniform distribution – as explained in section 7.4.2.

PaX is much better than Linux in 32 bits, but quite similar in 64 bits.

ASLR-NG: In 32 bits, libraries and mmapped objects have almost 20 bits of entropy, which is comparable with the minimum randomised objects in 64-bit Linux (vDSO and ARGV). Because of the small VM space in 32 bits the entropy is intrinsically limited, but thanks to the ability of the ASLR-NG to use the full address range to allocate any object, it increases entropy by up to 20 more bits than Linux and 12 more than PaX. Although ASLR-NG provides the highest randomisation for huge pages, the alignment constraint (which resets the lowest 22 bits) only leaves the possibility of randomising the highest 10 bits.

In 64 bits, ASLR-NG provides up to 15 more bits than Linux and 14 more than PaX. Regarding huge pages, Linux and PaX have 1 million possible places to load huge pages compared with the 67 million of ASLR-NG. This increment in entropy, jointly with the specific-zone for huge pages, increases the cost for an attacker to guess where they are placed and at the same time prevents using them in correlation attacks.

Hence, ASLR-NG outperforms Linux and PaX ASLR in both 32- and 64-bit systems.

7.7.4 Correlation in ASLR-NG

ASLR-NG addresses correlation weakness by randomising objects and specific-zones independently. Obviously, all the objects allocated in the same specific-zone are correlated together, but they are uncorrelated in relation to other specific-zones or objects.

The concentrated mode, by definition, is fully correlated to provide a compact layout to systems with low resources. The conservative mode is close to Linux and PaX but prevents using the stack, executable and heap in correlation attacks.

In extended mode, ASLR-NG extends the conservative mode by five specific-zones to isolate objects of different criticality levels. The paranoid mode goes a step beyond, though, by removing the correlation between all pairs of objects (no specific-zones are created), but as far as we know, exploiting the correlation between objects in the same category is not useful. Typically, a single library contains enough gadgets to build ROP exploits, and so it is not necessary to de-randomise other libraries.

7.8 Conclusions and future work

In this paper, we have shown that Linux and PaX ASLRs are weak, and four weaknesses have been presented to demonstrate this point. We developed **ASLRA**, a tool to analyse ASLR implementation, which helped us to identify and evaluate these weaknesses, and we also designed ASLR-NG, a realistic ASLR which includes novel randomisation forms and maximises entropy. We have shown the effectiveness of the **ASLR-NG** against multiple attack vectors (absolute and correlated attacks) and have developed a working prototype in Linux 4.1 which demonstrates the feasibility of ASLR-NG.

The main features of ASLR-NG are:

- Uses full memory space to randomise objects, which in turn provides maximum entropy.
- A novel solution for reducing fragmentation, without reducing entropy.
- Objects containing sensitive information are automatically isolated.
- Sequentially loaded libraries are randomised.
- It provides strong protection against absolute and correlation attacks, which effectively removes the four weaknesses previously identified.

Although in 64-bit systems ASLR-NG provides very strong protection, in 32-bits the improvements made by ASLR-NG with respect to current designs is more appealing because of the necessity of additional entropy.

ASLR-NG outperforms the PaX ASLR in all aspects, which as far as we know is currently the best (most secure) design and implementation of ASLR.

Part III

Diversification Through Emulation

Chapter 8

DRITAE: Diversified Replication Infrastructure Through Architecture Emulation

Since memory error exploitation usually relies on highly specific processor characteristics, the same exploitation rarely works on different hardware architectures. This chapter proposes a novel strategy to thwart memory error exploitation by dynamically changing, upon crash detection, the variant executing the networking server.

Required software diversification among variants is automatically generated using off-the-shelf cross-compilation suites, whereas hardware diversification relies on efficient processor emulation tools.

Contents

8.1 Introduction

Computer systems are under constant threat by hackers attempting to seize unauthorised control for malicious ends. Memory errors have been around for over 30 years and, despite research and development efforts carried out by academia and industry, they are still included in the CWE SANS top 25 list of the most dangerous software errors [3]. Classically, they were exploited by pursuing the remote injection of binary code into the target application's memory and then diverting the control flow toward the injected code. Today, memory error exploitation has evolved into code-reuse attacks, where no malicious code is injected and legitimate code is reused for malicious purposes [15]. Interested readers can find in [71] a very detailed analysis of the past, present and future of memory errors, and as the authors conclude, "they still represent a threat, undermining the security of our systems."

Different approaches have been proposed and developed so far, in order to eradicate or mitigate memory errors and their exploitation. The use of safe languages [72] is maybe the most effective approach, since it removes memory error vulnerabilities entirely. The idea consists of imposing stronger memory models on programming languages, in order to increase their safety. Other effective strategies for fighting against memory error infiltration rely on bounds checkers [73], which audit programs execution for out-of-bounds accesses, or the deployment of countermeasures to prevent overwriting memory locations [32], detecting code injections in the early stages [74] or preventing attackers from finding, using or executing injected code [75]. Commonly, these techniques rely on keeping secret key information required by attackers to break the system's protection. It must be also noted that all of them lead protected systems or applications to crash in the case of memory errors. Although not perfect, these solutions have shown their usefulness in greatly reducing the success rates of attackers, which is why they are nowadays incorporated in most computer systems [76].

The decision to abort an application, to hinder an attack, can be called into into question, however, especially in the context of networking business-critical services. It has been reported that the cost of one hour of downtime for an airline reservation centre is about $89,000, while for eBay this figure is about $225,000 and in the case of credit card authorisation the cost shoots up to $6,450,000 [77]. In order to deal with this unavailability problem, modern server-oriented architectures commonly make use of process-based abstractions as error containment regions [78]. As a result, and despite the crashing of a concrete process serving a particular request, the server may continue to process ongoing and new requests.

Another pending issue in existing crash-based protection techniques relates to their inability to keep the confidentiality of secret key information in the presence of brute force attacks. Depending on the protection technique and how the targeted application is internally ‘architectured’, the number of tries or guesses required to establish the secret varies; for instance, it takes around 216 seconds to bypass the address space layout randomisation mechanism included in an Apache server running on Linux [14]. Unfortunately, this is too short a time to enable system administrators to deploy any effective countermeasure.

In order to face the problem of brute force attacks, PaX developers group recommendation relies on combining existing protection techniques with a “crash detection and reaction mechanism” [31]. This approach could be applied to any protection technique that causes the attack process to crash (wrong guess in relation to secret key information), thus becoming detectable. As already mentioned, nowadays very limited actions are usually taken when brute force attacks are detected: either the service is shutdown, accompanied by the subsequent economic cost, or it keeps running and an alert is issued to administrators, who may not be able to intervene fast enough to prevent a successful intrusion.

Diversification is an approach with a great deal of potential to build effective defences against attacks in general and brute force attacks against Web servers in particular. Applied to systems development, diversification can be seen as a system (or application) with at least two *variants*, plus a decider which monitors results from variant execution, in order to make decisions affecting their execution [79]. These variants are different versions of the same application (coming from different designs and/or implementations) that, although being different, behave as expected from their specification, i.e. they provide the same service as perceived by users. However, differences existing among variants lead them to exhibit dissimilar sets of vulnerabilities – and thus different degrees of sensitivity – against accidental and malicious faults.

The proposal presented in this paper builds on the principle that the exploitation of memory errors relies on highly specific processor characteristics, so the same procedure rarely works on different hardware architectures. Obviously, diversifying the hardware also means diversifying the considered software. Software diversification, i.e. the production of server variants, will be achieved by using off-the-shelf, cross-compilation suites, whereas hardware diversification relies on the emulation of different processor architectures. In this way, some vulnerabilities which manifest in a given architecture could be removed just by changing the execution platform to another particular architecture in which existing software faults no longer constitute a form of vulnerability. So, basically, a variant replacement policy is deployed when detecting a process crash issued as a result of memory errors. The approach can be combined seamlessly with existing protection techniques to complement a highly secure mechanism in the fight against memory error exploitation.

The key contributions of this paper are as follows:

1. It proposes a multi-architecture variant system running on a single platform, taking advantage of improvements in emulation support.
2. It employs off-the-shelf, cross-compilation toolchains as a diversification technique.
3. A novel recovery strategy, after an attack attempt, is developed that maintains service continuity while invalidating brute force attacks and preventing the manifestation of some accidental faults.
4. The paper demonstrates, through two case studies, the effectiveness and portability of the technique as well as low implementation costs thanks to the reuse of already existing tools.

The rest of this paper describes this proposal in detail. Section 8.2 provides the background required to understand the problem tackled by this solution. Section 8.3 details the DRITAE approach, whereas section 8.4 reports all practical aspects related to its deployment on a real Web server running on two different platforms. The results produced by the evaluation of the developed prototypes are presented in section 8.5 and discussed in section 8.6. Finally, conclusions are provided in section 8.7.

8.2 Background and challenges

This section describes a number of vulnerabilities that commonly lead to memory errors, already existing mechanisms developed to cope with this problem and attacks that could be used to bypass these mechanisms and successfully exploit existing vulnerabilities in the context of networking servers. Finally, an identification of the common characteristics of presented vulnerabilities and attacks paves the way to defining a new architecture for memory error prevention. Figure 8.1 maps the various notions introduced in this section in relation to the well-known AVI (Attack + Vulnerability $\rightarrow$ Intrusion) model [80].

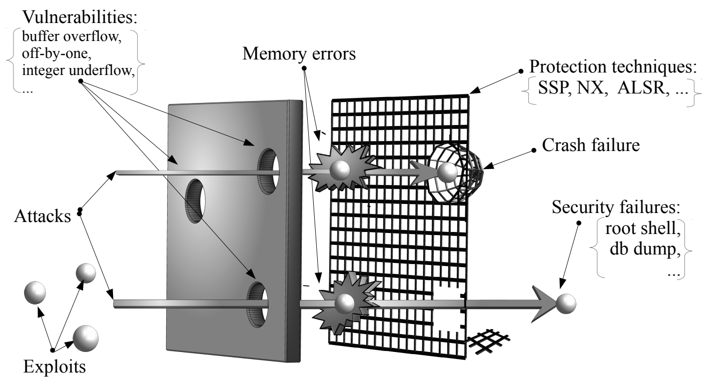


FIGURE 8.1: Mapping memory errors to the AVI model

8.2.1 Memory errors

Memory errors usually derive from the exploitation of vulnerabilities (depicted as a wall with holes in Figure 8.1) existing in a given application caused by software faults introduced during its implementation. The most common software faults leading to memory errors include off-by-one, integer and buffer overflow vulnerabilities.

Off-by-one [81] vulnerabilities write one byte outside the bounds of allocated memory, and they are often related to iterative loops iterating once too often or common string functions incorrectly terminating strings. For instance, the bug reported by Frank Bussed [82] in the `libpng` library allowed remote attackers to crash an application via a crafted PNG image that triggered an out-of-bounds read during the copying of error message data.

Integer vulnerabilities [83] are usually caused by an integer exceeding its maximum or minimum boundary values. They can be used to bypass size checks or to allocate buffers to a size too small to contain the data copied into them. A recent bug discovered in the `libpng` library did not properly handle certain malformed PNG images [84], and it therefore allowed remote attackers to overwrite memory with an arbitrary amount of data, and possibly have other unspecified impacts, via a crafted PNG image. Vendors affected included Apple, Debian GNU/Linux, Fedora, Gentoo, Google, Novell, Ubuntu and SUSE, among others.

Buffer overflows [11] are caused by overrunning the buffer's boundary while writing data into said buffer, which allows attackers to overwrite data that controls the program execution path and hijack control of the program to execute the attacker's code instead of the process code. A recent stack-based buffer overflow example can be found in the `cbtls_verify` function in FreeRADIUS, which causes server crashes and possibly executes arbitrary code via a long 'not after' time stamp in a client certificate [85].

Over the last decade, different techniques have been developed to prevent attackers from successfully exploiting these vulnerabilities, thus reducing the chance of causing memory errors.

8.2.2 Protection mechanisms

The most effective protection techniques commonly used nowadays to fight against memory errors, represented as a grid in Figure 8.1, comprise address-space layout randomisation, stack-smashing protection, a non-executable bit and instruction set randomisation.

Address space layout randomisation (ASLR) [31]. Whenever a new process is loaded in main memory, the operating system loads different areas of the process (code, data, heap, stack, etc.) in random positions in the virtual memory space of the process. Attacks relying on knowing precisely the absolute address of a library function, such as `ret2libc`, or the already injected shell code, are very likely to crash the process, thereby preventing successful intrusion.

Stack-smashing protection (SSP) [17]. A random value, commonly known as a *canary*, is placed on the stack, just below the saved registers from the function prologue. This value is checked at the end of the function, before returning, and the program aborts if the stored canary does not match its initial value. Any attempt to overwrite the saved return address on the stack will also overwrite the canary and lead to a process crash, to prevent intrusion.

The non-executable bit (NX), or "W$\oplus$X" [32]. Memory areas (pages) of the process not containing code are marked as non-executable, so they cannot be written. On the other hand, those areas containing data are marked as just writeable, so they cannot be executed. Processors must provide hardware support to check for this policy when fetching instructions from main memory. Even if an attacker successfully injects code into a writeable (not executable) memory region, any attempt to execute this code will lead to a process crash.

Instruction set randomisation (ISR) [86]. This technique randomly modifies the instructions (code) of the process, so they must be properly decoded before being effectively executed by the processor. Successful binary code injection attacks will crash the process, as decoding the injected

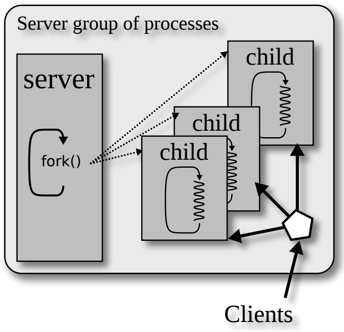


FIGURE 8.2: Multi-process model for server architectures

code will not produce correct instructions. Unlike previous techniques, ISR is less commonly used.

Despite the high levels of protection provided by these techniques, their effectiveness is reduced significantly in the case of networking servers. Typically, the implementation of this type of server is crash-resilient (see crash failure in Figure 8.1), which increases the availability of the provided service; however, this makes them very sensitive to brute force attacks (note the grid hole leading to security failures in Figure 8.1). The next section focuses on this problem.

8.2.3 Networking server weakness

Traditionally, networking server architectures [87] have come in two main packages: thread-based and process-based architectures.

Multi-threaded architectures associate incoming connections with separate lightweight threads. These threads share the same space address – and thus global variables and states. Furthermore, they require small amounts of memory and provide fast inter-thread communication and response times, thereby making them suitable for high-performance servers. However, memory errors on one thread may corrupt the memory of any other thread, resulting in compromised threads accessing sensitive data from the rest of the threads.

The compartmentalisation philosophy promoted in security manuals better fits multi-process architectures. Incoming connections are handled by separate child processes which are *forked* (created) by making an exact copy of the memory segments of the parent process in a separate address space (see Figure 8.2). Although performance suffers from the effects of larger memory footprints and heavyweight structures, these architectures are more suitable for, and typically used in, highly secure servers [87].

Nevertheless, the common operation of multi-process servers makes them vulnerable to different attacks because, since all the children have the same secrets (ASLR offset, canary value, etc.) as the parent, a brute force attack can be created.

ASLR provides little benefit for 32-bit systems, as there are only 8 random bits for mmapped areas, and the secret can be guessed by brute force in a matter of minutes [14].

Applications protected with SSP are vulnerable to buffer overflows in the heap or via function overwrites and/or brute force attacks [12]. The most dangerous vulnerabilities are those allowing a 'byte-for-byte' brute force attack, which will compromise the system through 1024 attempts (32-bit machine), such as the latest pre-auth ProFTPd bug [38].

The NX technique is easily bypassed by overwriting the return address on the call stack, so instead of returning into code located within the stack, it will return into a memory area occupied by a dynamic library [36]. Typically, the `libc` shared library is used, as it is always linked to the program and provides useful calls to an attacker (such as `system("/bin/sh")` to get a shell).

ISR is vulnerable to brute force attacks, like SPP, and also to attacks that only modify the contents of variables in the stack or the heap, which causes control flow changes or the logical operation of the program [86].

Although several techniques have been proposed to date, to prevent the successful exploitation of memory errors, the truth is that all of them can be bypassed one way or another. The next section focuses on how to complement these mechanisms with an approach that can be used to improve their resilience against brute force attacks.

8.3 DRITAE architecture

The core idea behind DRITAE consists of having the same application compiled for different processors and replacing the executable process when an error is detected. Each variant is executed in sequential order on the same host by a fast processor emulator. In the case of a malicious attack, since code execution is highly processor-dependent, changing the processor that runs the application greatly hinders attack success. The DRITAE architecture has the following elements:

1. A set of cross-compiler suites for creating the set of variants.

2. A set of emulators for running the variants.
3. An error detection mechanism which triggers variant replacement.
4. A recovery strategy which selects the variant that will be used once an error has been detected.

This approach maintains service continuity while trying to fix a fault (in the case that the fault does not manifest in one of the variants) or difficulties caused by a malicious attack.

8.3.1 Creation of variants

Many diversification techniques are based on compiler or linker customisations for the automatic generation of variants [88]. Although eliminating the need to manually rewrite the source code for diversification, deploying the required customisations on different compilers/linkers, or introducing new modifications, is costly and prone to new errors.

The proposed approach (see Figure 8.3) promotes the use of already existing cross-compilers to generate variants, one for each target architecture, in an easy and effective way. Cross-compiler toolchains provide the set of utilities (compiler, linker, support libraries and debugger) required to build binary code for a platform other than the one running the toolchain. For instance, the GNU cross-compiling platform toolchain is a highly portable widespread suite which is able to generate code for almost all 32- and 64-bit existing processors.

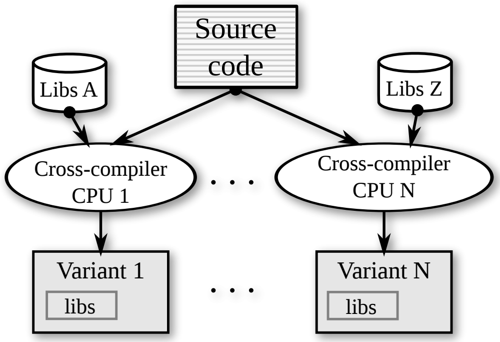


Figure 8.3: Variant generation

Just by compiling the application source code for different target processors, the particular architecture of each processor will provide variants with a different:

Endianness and instruction set, so raw data and machine code injected by attackers will be differently interpreted;

Register set, thus changing the stack layout (on non-orthogonal architectures);

Data and code alignment, so unaligned instructions and word data type will raise an exception;

Address layout, which results in different positions for functions and main data structures according to resulting code size and data layout and

Compiler optimisations, some generic and some processor-specific, resulting in register allocation, instruction reordering or function reordering.

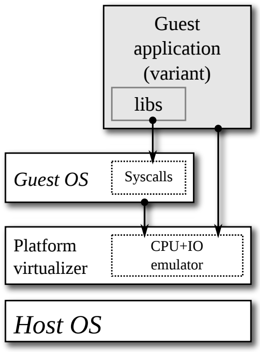
Furthermore, most applications use the services provided by one or more libraries which are linked during the compilation process. For instance, the standard C library provides an interface to the operating system (system calls), basic algorithms (string manipulation, maths function, sorting, etc.), type definition, etc. As several implementations of the C library exist to attain different goals, such as licence issues, small memory footprint, better portability or multi-thread support, among others, by linking variants with different libraries it is possible to i) gain a higher degree of diversification among them and ii) get rid of specific software faults that are not present in some libraries.

This form of binary diversification preserves the semantic behaviour of each variant, it is easy to implement, because of the reuse of widely available and tested software, and it provides strong differentiation between resulting binaries.

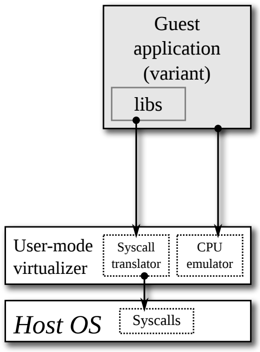
8.3.2 Execution of Variants

Variants require a proper execution environment, including the operating system API, a system calls convention, a processor instruction set and the executable file format, to be run. The native variant, i.e. one compiled for the physical processor and operating system hosting the server, will run in the native execution environment. However, as the rest of the variants have been built for different processors, it is necessary to create a virtual execution environment in which to run them all.

Nowadays, there are two different virtualisation[1] solutions (see Figures 8.4) that can be used to build a complete execution environment: **i) platform emulation**, where the emulator provides virtual hardware to execute the guest operating system managing the guest application, and **ii) user mode emulation**, where the emulator provides both processor virtualisation and operating system services, translating guest system calls into host system calls that are forwarded to the host operating system.



(a) Platform



(b) User-mode

FIGURE 8.4: Approaches to virtualisation

User-mode emulation is a less common form of emulation but offers better performance, since the operating system code is directly executed by the host processor. The emulator loads the guest-executable code into its process memory space, and then it is dynamically translated into host native code and system calls are emulated (converted from a guest format to the host and back). Conceptually, user-mode emulation is very close to the Java Run-time Environment (JRE), which is a software emulator employed for running the Java virtual machine specification. The main difference between user-mode emulation and JRE is that the former emulates real processors and real operating systems, while the latter emulates the Java virtual machine specification.

According to these benefits, this proposal promotes the use of user-mode emulation to create the execution environment required for each variant. Thus, variants should be compiled for the same operating system of the host machine (or a compatible counterpart).

8.3.3 Memory error detection

DRITAE architecture relies on existing protection mechanisms (SSP, ASLR, etc.) to crash the compromised process. A monitor will be in charge of detecting these crash-related events and triggering the established variant replacement strategy according to the defined security policy.

It must be noted that, although those techniques were initially developed to face malicious faults, they also provide good coverage for accidental faults, like wild pointers. Accordingly, the accidental activation of software faults leading to memory errors will also crash the process and give the system a chance to deal with them accordingly.

The precise diagnosis of whether the problem is related to an accidental or malicious fault and its precise origin (kind of attack), to define a more specific reaction, is still an issue for further research.

8.3.4 Variant replacement strategy

The widely used multi-process architecture of networking servers provides an ideal scenario for deploying different security policies for variant replacement upon the detection of memory errors. The proposed policy consists of three successive stages (see Figure 8.5):

1. High-performance service.
2. Fault avoidance.
3. Confuse the attacker.

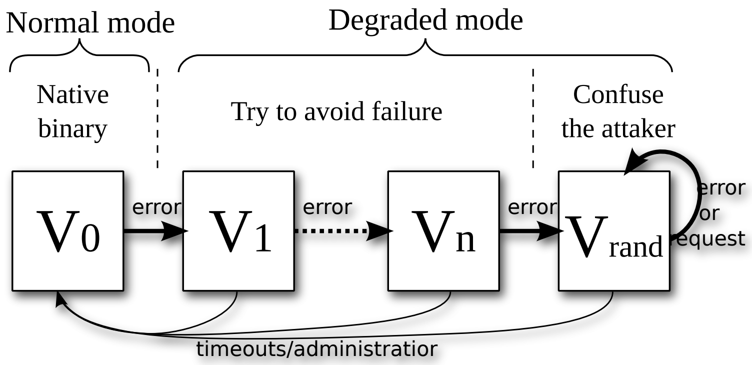


FIGURE 8.5: Variants replacements policy

8.3.4.1 Stage 1: High-performance service

Initially, the service is provided by the native variant, directly running on the host computer, and thus it is **free from any overhead due to virtualisation**. When detecting a crash, the system will enter into a fault avoidance mode.

8.3.4.2 Stage 2: Fault avoidance

In this second stage, the system assumes that an attack, or an accidental software fault, exists which could again cause a crash on another child. In order to try to hinder or even prevent the successful exploitation of the memory error, the next variant to be executed will be selected from among those with more architectural differences with respect to the previously selected one. For instance, a buffer overflow by one byte is likely to cause an error on the i386 architecture but not on the SPARC one, due to the different way the processor registers are managed, Apache chunked-encoding is exploitable on 32-bit processors but not on 64-bit Unix platforms [89] or a busybox integer overflow [90] only affects big endian systems. After changing the variant, the service will be running in a performance-degraded mode due to processor emulation overheads. This performance penalty also hinders attackers by slowing down the attacks, which gives valuable time to administrators to fix the problem. If no new crashes occur during a given period, the system could automatically revert to the native mode to increase its performance, or it could require an explicit command from the system administrator to do so. In the case of new process crashes, the system will keep changing from one variant to another until all of them have been tried. When no new variants are available, the system will assume it is under a brute force attack and the fault is manifesting on all variants, which requires a more aggressive replacement policy.

8.3.4.3 Stage 3: Confuse the attacker

The third stage focuses on confusing a possible attacker, in order to reduce as much as possible information that could be retrieved from unsuccessful exploitation attempts. In the case of keeping a sequential variant replacement, although highly difficult and time costly, expert attackers may finally guess the processor architecture of some variants and could develop some form of exploitation to compromise the system. Accordingly, the proposed policy relies on randomly selecting the next variant to be executed, which makes it more difficult to launch a brute force attack. The policy to revert to native mode is the same as in the second stage.

Obviously, different policies should be defined according to the particular needs and resources available to each server, such existing variants or diagnosis and detection capabilities, so they vary from one particular case to another.

Since each request is redirected to a different variant independently, regardless or whether or not they crash, **brute force attacks are no longer possible**.

8.4 Case study: Web server

In order to prove the feasibility and portability of the DRITAE, an HTTP server running on a computer and an smartphone has been chosen as a case study.

The first hardware platform, selected as a representative of a common platform for Web servers, consists of a computer running an Ubuntu 12.04.1 LTS operating system. The PC is equipped with an x86_64 Intel Core i3-370M CPU, clocked at 2.4 GHz and with 3072 MB RAM.

The second target platform, selected to show the portability of DRITAE, even to devices with limited resources, is a Samsung Galaxy S smartphone running the Android 2.3.6 operating system. The smartphone is equipped with a 1 GHz ARM Cortex A8 processor with 512 MB RAM and a PowerVR SGX 540 GPU.

The busybox-httpd application [91] has been selected to provide the required networking service. It is a complex, fully featured application, widely used on many platforms, including smartphones, routers and media players, running in a variety of POSIX environments such as Linux, Android and FreeBSD, among others. The busybox application consists of a single executable file that can be customised to provide a subset of over 200 utilities specified in the Single Unix Specification (SIS) plus many others that a user would expect to see on a Linux system, including the *httpd* Web server considered in this case study.

The following sections describe in detail the particular instantiation of all the elements required to deploy DRITAE, including variants created using cross-compilation, processor emulation support, the detection mechanism, the recovery strategy deployed for variant replacement procedure and the security policy. It must be noted that the following implementation has been seamlessly applied to both considered hardware platforms (PC and smartphone) albeit with minor changes. An overview of the concrete prototype developed for the smartphone is depicted in Figure 8.6.

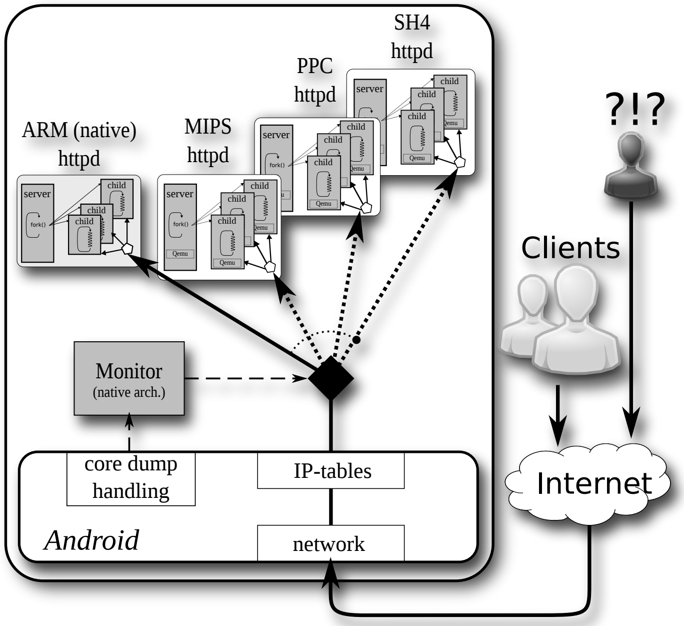


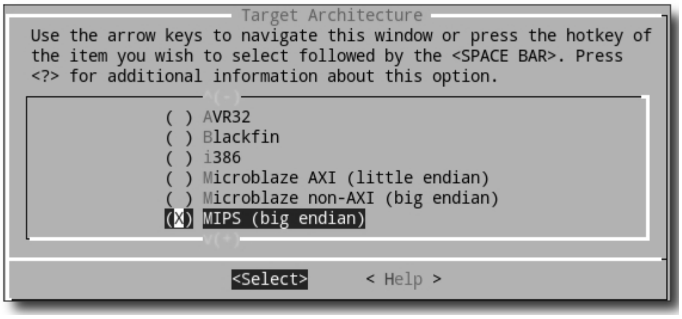
Figure 8.6: System prototype overview for the smartphone

8.4.1 Building cross-compilers

In order to build the binary images (*httpd* variants) for each target architecture, a cross-compiling toolchain suite is required for each of them. One possibility consists in downloading pre-compiled versions from different projects or providers, which would not be flexible enough to achieve the desired level of diversification. On the other hand, the required toolchain suites could be built from the source code of each element (compiler, linker, library and debugger) to acquire greater controllability when creating variants.

Although building a toolchain is quite tricky, due to the strong dependence among elements, thanks to the buildroot project[1] it is possible to build (and customise) a GNU toolchain very easily. Buildroot uses the same source code configuration tools as the Linux kernel, commonly known as *menuconfig*, which presents a simple menu interface (see Figure 8.7) guiding the user to configure code features while avoiding conflicting or incompatible options.

[1]Buildroot project: http://buildroot.net



```
Target Architecture
Use the arrow keys to navigate this window or press the hotkey of
the item you wish to select followed by the <SPACE BAR>. Press
<?> for additional information about this option.
                ^(-)
        ( ) AVR32
        ( ) Blackfin
        ( ) i386
        ( ) Microblaze AXI (little endian)
        ( ) Microblaze non-AXI (big endian)
        (X) MIPS (big endian)
                v(+)

             <Select>      < Help >
```

FIGURE 8.7: Buildroot configuration menu interface

Buildroot V2012.05 was selected for this case study and, according to the target architectures, generation parameters were customised to build the required cross-compilers: i) according to the host operating system the selected *Kernel Headers* were "Linux 3.2.x kernel headers", ii) considered *Target Architecture* include the native ones, "x86_64" and "ARM", and some others with very different architectures, like "SPARC", "i386", "SH4", "MIPS", and "PowerPC", iii) the *uClibc C* library version was "uClibc 0.9.32.x", and iv) as the selected version of the processor emulator does not support "Native POSIX Threading (NPTL)" for all architectures, the "linuxthreads (stable/old)" *thread library implementation* was used. The remaining options were configured to be as similar as possible, in order to perform an accurate comparative analysis.

8.4.2 Qemu emulator

Qemu [92] is a generic, open source and fast machine emulator and virtualiser that uses a portable dynamic translator for various target CPU architectures. Just as an example of its high quality and popularity, it is the base of the Android emulator currently distributed within the Android SDK. Besides the standard CPU emulation mode, Qemu implements user-mode emulation, which can run a single program (process) in a complete virtualised environment and constitutes the core of the DRITAE approach, as described in section 8.3.2. User mode emulation is not limited to statically compiled binaries, as it can also load dynamic libraries, thereby enabling the direct execution (emulation) of most existing applications.

Combining user mode emulation with the Linux capability to run arbitrary executables (called `binfmt_misc`), it is possible to run a guest binary executable (variant) as if it were a native one. Note that it is the operating system kernel and not a module of the command interpreter or another user application which interprets the executable format. In fact, all variants (regardless of the target processor) use the same operating system and can access the same directory hierarchy and network interfaces. Thanks to this form of emulation, it is not necessary to set up a complete virtual platform, and variants are transparently and more efficiently executed as if they were native processes.

8.4.3 Detecting crashes

The *core dump* facility of Linux (also available on many operating systems) was selected as a suitable tool for detecting the abnormal termination of variants. Core dumps are triggered by different signals (see Table 8.1). The operating system infrastructure for dumping process core images (`/proc/sys/kernel/core_pattern` file) provides, since Linux 2.6, facilities to send the core image to a crash reporter program along with command-line information about the crashed process.

Although core dumps provide lots of useful information for diagnosing and debugging programming errors, the proposed approach just requires a notification that the process has crashed, regardless of its cause (as previously mentioned, this could be an enhancement requiring further research). The `core_pattern` file is configured (see Listing 8.1) to call a tiny shell script `log_crashes.sh` (see Listing 8.2) whenever any system process crashes. This script will receive the PID of the crashed process (`%p`), the triggering signal (`%s`), the executable filename name (`%e`) and the time of dump

Signal	**Description**
SIGQUIT	Quit from keyboard
SIGILL	Illegal instruction
SIGABRT	Abort signal from abort(3)
SIGFPE	Floating point exception
SIGSEGV	Invalid memory reference
SIGBUS	Bus error (bad memory access)
SIGTRAP	Trace/breakpoint trap
SIGXCPU	CPU time limit exceeded
SIGXFSZ	File size limit exceeded
SIGIOT	IOT trap. A synonym for SIGABRT

TABLE 8.1: Signals leading to a core dump

(`%t`), following which it will append a single line containing the received information into a file named `/var/log/m/crashes.log`.

```
echo "|/var/log/m/log_crashes.sh %p %s %e %t"
   >/proc/sys/kernel/core_pattern
```

Listing 8.1: Command for configuring `core_pattern`

The Linux *inotify* mechanism, which provides an efficient file system events monitoring service, has been used to monitor when new entries are written in the `crashes.log` file. This allows one to read new entries from the file and consider only those caused by variant processes without overheads. Note that the monitor is compiled for the native architecture and will be blocked (inactivated) until a new entry is written.

```
#!/bin/bash

pid=$1
signal=$2
ex_name=$3
time=$4

fcrash="/var/log/m/crashes.log"

echo "[$pid] [$signal] [$ex_name]  [$time]" >> $fcrash
```

Listing 8.2: Saving core dump information into `crashes.log`

As the monitor is a critical component in this approach, it has been designed so as to be simple and small, with the aim of minimising the probability of introducing design or software faults. Another requisite was to isolate the monitor from the application so that attackers would not be aware of its presence and benefit from any communication channel to interfere with it.

Contrary to other protection solutions, the monitor does not act as a barrier between clients/attackers and servers or add new code or features which attackers could exploit. The kernel facilities used to handle core files enable the immediate detection of crashed variants, without modifications to either the variants' code or their configuration. Also, it is important to note that it is a one-way communication channel with a very limited and simple interface, which subsequently makes it extremely difficult to successfully attack the monitor through this channel.

8.4.4 Alternating among variants

Once the crash has been detected, the current variant shall be replaced by another one according to the established replacement policy. The most straightforward solution would be to stop (kill) all the processes (in the case of a multi-process server) of the current variant and start up the next one. However, this solution presents two important drawbacks: i) the failure of a single server process serving a particular client is propagated to the rest of the processes, and thus all ongoing connections are affected by the failure, and ii) the service is unavailable until the next variant is up and running.

A less drastic solution can be implemented using kernel firewall facilities, known as *iptables*, which allows a system administrator to customise the tables provided by the Linux kernel firewall and the chains and rules it stores. Using *iptables*, active connections are preserved for those server processes that are working properly and can redirect the new connections (clients) to the new selected variant, thus solving any *μ-denial-of-service (μ-DoS)* or temporal service unavailability problems.

Following this approach, all variants are created and started as if they were the actual server. Each variant is configured to listen for connections on different ports (other than the external server port), which are blocked using *iptables* to prevent external connections from directly accessing the variants. The internal firewall is then configured to redirect incoming connections from the service port to the active port of the current variant. When a process of the current variant crashes, the policy of the recovery strategy is applied to decide which will be the next variant, following which the service port is redirected to the next variant port. A sample *iptables* rule implementing this approach is shown in Listing 8.3.

Following the isolation design principle of the monitor, the variant selection procedure, implemented using the *iptables* facility, is an *indirect* mechanism that benefits from the kernel IP routing tables. Variants are not aware of the presence of the monitor and they are not modified in any way. In this case, there is no communication channel through which attackers can reach the monitor by accessing variants.

```
iptables -A PREROUTING -t nat -p tcp
          --dport [service-port]
          -j REDIRECT --to-ports [variant-port]
```

LISTING 8.3: Firewall configuration to change the active variant

In this prototype, all variants are simultaneously launched when the service is started. More advanced replacement and recovery policies enabled by the kernel firewall facility are discussed in section 8.6.

8.5 Experimentation and results

In order to assess the effectiveness of DRITAE, the prototypes considered in the case study have been exposed to a number of exploitable vulnerabilities, leading to memory errors. Results show the importance of properly selecting hardware architectures to prevent the further exploitation of existing vulnerabilities, either because software faults no longer lead to memory errors or because brute force attacks get confused. Finally, the temporal and spatial overheads induced by the solution are analysed.

8.5.1 Fault manifestation

The exploitation of software faults leading to memory errors may present a different manifestation according to the variant under execution, due to its particular processor architecture. To illustrate this point, a buffer overflow fault has been injected into the busybox *httpd* Web server (see Listing 8.4) whereas, for the sake of clarity, off-by-one and integer underflow faults have been manually injected into a standalone program (see Listing 8.5).

```
static int get_line(void) {
  int count = 0;
  char c;
  char buffer[256];  // Injected code
        ...
        ...
  strcpy(buffer, iobuf);  // Injected code
  return count;
}
```

Listing 8.4: Buffer overflow fault injected in busybox-httpd

The code injected into the `httpd.c` file, in the function `get_line(void)` (see Listing 8.4), constitutes a typical buffer overflow, similar to the one found in Oracle 9 [93]. The URL of the HTTP request will be copied into the added buffer but, as there is no length check, long URLs will flood the buffer and cause a memory error. HTTP requests with increasing URL lengths have been tested for each variant, and Table 8.2 lists the minimum length required to crash the process. Results show that the SPARC architecture is – by far – the most robust form of defence against that particular fault, so it could be a good choice to prevent problems derived from buffer overflows.

The `offByOne()` function from Listing 8.5 line 1, inspired in one affecting an FTP server [94], includes an offset-by-one software fault, since arguments with a length of 128 will cause the `strcpy()` function to overflow the buffer by just one byte (the appended '\0' char). Table 8.2 lists whether this fault

```
void offByOne(char *arg) {
  char buffer[128];
  if(strlen(arg)>128) {
    printf("Overflow\n");
    exit(0);
  }
  strcpy(buffer, arg);
}

void intUnderflow(unsigned int len, char *src){
  unsigned int size;
  size = len - 2;
  char *comment = (char *) malloc (size + 1);
  memcpy (comment, src, size);
}
```

Listing 8.5: Code for off-by-one and integer underflow

manifests for two different variants created for each considered architecture: one with the commonly used `-O2` optimisation flag and the other with no optimisations (`-O0`). The SPARC is again the most robust architecture, as the fault does not crash the process, regardless of the selected optimisation flags, whereas the SH4 architecture always crashes. It must be noted the negative influence of compiler optimisations that, in general, produce less robust variants.

The integer underflow software fault has been tested using a real-world vulnerability [95] in a JPEG processing code. The code of the faulty `intUnderflow()` function is shown on Listing 8.5 line 10. When `1` is passed as the first parameter, the `size` variable has the value `-1`, which is interpreted as a large positive value (`0xFFffFFff`) in the third parameter of the `memcpy` function (line 14) in a 32-bit architecture. This incorrect value is then used to perform a memory copy into the buffer reserved by the

Variant	**Off-by-one** `-O2` flag	`-O0` flag	**Integer underflow**	**Buffer overflow** bytes to crash
x86_64	Crash	No crash	Crash	68
i386	Crash	No crash	Crash	250
ARM	Crash	No crash	No crash	62
MIPSEL	No crash	Crash	No crash	258
SPARC	No crash	No crash	Crash	1234
SH4	Crash	Crash	Crash	50
PPC	No crash	No crash	Crash	70

Table 8.2: Number of attempts on different software faults.

previous `malloc()`. The behaviour of a `malloc()` request for zero bytes is implementation-dependent (some implementations return NULL, while others return a pointer to the heap area). As shown in Table 8.2, only the ARM and MIPSEL variants prevent the process from crashing.

8.5.2 Protection against attacks

The basic idea behind brute force attacks consists of making continuous requests, trying all the possible values of an unknown *secret* (a memory address or a random value, for instance), until the right value is found. On systems equipped with ASLR, NX, and stack protector techniques, the typical steps to build an attack are as follows:

1. Find out the offset to the canary on the stack. It can be estimated accurately from the application image (we assume that the attacker has got it), but it is common to verify the offset by testing sequentially the position of the canary.
2. Find out the value of the canary (using brute force against the target).
3. Build the ROP[2] sequence (based on the ELF). For simplicity, we will assume that ROP gadgets come from the libc.
4. Find out the entry point of the ROP (using brute force against the target).

Depending on the kind of error, not all of these steps are required; for example, a memory error in the data segment can be exploited without knowing the canary.

The final exploit string must have the correct values for all of the following elements: canary value and offset, ROP sequence and entry point. Those parts of the exploit that are not known by the attacker can be obtained using brute force, by building partial exploitation up to the values that are already known and then testing only the value that is missing. Any incorrect value (or a partial string) is detected by the protection mechanisms and the application is crashed. Since the protection mechanisms are applied sequentially, the attacker is able to build a partial exploitation which only affects one of these protections. Once the protection is bypassed, the next one can be addressed.

Our solution nullifies the possibility of building exploitations in this way, because the following basic assumptions no longer hold:

- The active server target is not always the same, and so a fault cannot be unequivocally interpreted as an incorrect value guess. Variants may crash not only due to a wrong guess, but also due to different memory layouts (invalid offsets) or endianness (invalid instruction/-data format), for instance.
- Some software faults may not be found in certain variants (as previously discussed), which will be interpreted by exploitations as a successful guess.
- Once an emulated variant is active, the performance of the server is degraded due to the emulation overhead, which plays against the attacker and increases the time required to guess the secret – in a similar way to the GRKERNSEC_BRUTE[96] option.

In [14], the authors show how the ASLR can be bypassed using a brute force attack to find out the correct address for a *return-to-libc*. The proposed exploitation succeeds, on average, in just 216 seconds. However, using our technique, the current variant will be replaced when the monitor detects a crash. In a variant with a different stack layout (the offset of the return address) the attacker will be overwriting a wrong one, and so the return address is not overwritten with the guessed value and consequently *the exploitation will never succeed.*

The most dangerous way to bypass the SSP mechanism is by employing the 'byte for byte' approach, whereas the most generic procedure consists in a generic brute force attack [20]. In the first case, if attackers overwrite individual bytes of the canary (secret), at most $\left(256 \cdot \frac{wordsize}{8}\right)$ tries (1024 attempts for a 32-bit machine) are required to guess the right value, which takes just a few seconds. For generic brute force attacks, attacks need to try all the possible values of the secret (2^{32} attempts at most for a 32-bit secret), returning the first one that does not cause a crash. This kind of attack will not be successful on the DRITAE architecture. After a variant is replaced, the address being overwritten by the attacker is not that belonging to the canary and thus *it will wrongly assume that the secret has been accurately guessed* if the process does not crash. Alternatively, if it crashes (not related to wrong guesses), *the attack will keep erroneously discarding possible values.* From this point on, the next steps of the attack are completely useless.

8.5.3 Spatial and temporal cost

In spite of the great security benefits provided by DRITAE, there is also a price to be paid in terms of spatial and temporal overheads, due to the execution of multiple virtual environments.

The spatial overhead refers to the amount of main memory consumed at run time for each variant when it is interpreted by the processor emulator (Qemu). This total amount of memory has three different components: i) the size of the executable image of the variant, which depends on the libraries selected for building the toolchain, the compiler optimisation flags and the code density of the related architecture, ii) the size of Qemu's memory translation cache, where application code is dynamically translated from the guest to the host architecture, and iii) the memory required by Qemu itself. Note that native variants have no memory overheads, since they are executed as if DRITAE has not been used.

The memory consumed by each variant has been estimated by means of a set of pages that are unique to a process ('unique set size' - USS) and been measured with the `smem(8)` tool. As both Qemu and variants are statically compiled there is no shared memory other than those pages shared between parent and children processes (all pages copied to children marked as copy-on-write). Table 8.3 summarises the average USS memory for each variant. Total memory used in the proposed implementation constitutes the memory used by all running variants. By default, all variants are launched but more conservative solutions can be used, where only the active and the next variant are ready (launched).

The temporal overhead is introduced as a result of the emulation support provided by Qemu when executing non-native variants. In the absence of crashes, the native variant is executed without any temporal overhead. This overhead has been estimated by means of the *Apache HTTP server benchmarking tool*, which was configured to perform 100 simultaneous requests for a total of 5,000 requests. Table 8.3 summarises the average latency and throughput (from the user view point) obtained by the considered *httpd* Web server when running the benchmark for each of the variants. It must be noted that requests to the computer were made locally, which could be considered as the best possible scenario, whereas requests to the smartphone were made remotely, thus occurring in all the penalties related to wireless networks, which could be considered as the worst possible scenario. The time required for the monitor to detect a process crash and configure the firewall is in the order of few microseconds, and so it is considered negligible and not included in the table in which latency is expressed in milliseconds.

		Memory (KB)		**Latency (ms)**		**Throughput (KB/s)**	
	Variant	PC	Phone	PC	Phone	PC	Phone
Native	x86_64	36	–	0.18	–	6290	–
	i386	32	–	0.26	–	4502	–
	ARM	–	32	–	9.8	–	120
Qemu	x86_64	408	400	2.23	55	529	21
	i386	416	336	2.35	52	503	22
	ARM	364	328	2.95	54	401	21
	MIPSEL	364	308	3.22	60	368	19
	PPC	404	346	8.02	118	147	10
	SH4	428	356	4.96	77	239	15
	SPARC	504	436	3.20	72	370	16

Table 8.3: Spatial and temporal overhead of the Web server.

8.6 Discussion

Successfully exploiting a memory error is not an easy task and cannot be achieved by just anyone. However, due to the proliferation of popular websites that act as a repository for existing attack strategies, even inexperienced hackers may have a chance to succeed. According to their knowledge and available resources, three basic types of attackers can be considered: script kiddies, black hat hackers and advanced persistent threat (APT) groups.

Most attacks come from script kiddies, or *skiddies*, who are non-expert users that simply download and use already existing strategies that are usually highly customised to target a very specific vulnerability on a given architecture. Thus, unless the attack succeeds at the very first try (quite unlikely, but there is always a chance), replacing the variant under attack will prevent this kind of user from successfully causing memory errors.

The knowledge and experience required to be able to exploit a software fault is only mastered by a few. Due to the complexity and the time required, black hat hackers are usually specialised on a given platform (operating system and architecture), and it is therefore very rare to find hackers able to target multi-architecture platforms like the one proposed in this work. Bypassing the DRITAE solution would required developing a meta-exploit, targeting multiple architectures, that could extract some valuable information from the running variant. As long as the next variant is randomly selected, it is rather unlikely that brute force attacks will succeed.

Finally, APT groups are made up of a set of people with both the capability/knowledge/resources and the intent to persistently and effectively break the security of a specific target. Even these groups, however, will be severely delayed in achieving a successful attack, not only by the complexity of developing a suitable exploit, but also due to the temporal overheads induced by the emulation process that greatly delays the attack. This gives administrators enough time to react and then deploy the countermeasures.

Following this line of thought, this technique should not be used as a stand-alone mechanism but be integrated into the security policy of the server. Whenever the server switches to degraded mode (a non-native variant is active), administrators shall be notified to deploy the most suitable countermeasures.

Typically, successful brute force attacks are launched by compromising a set of Internet-connected computers (*zombies*) that can collaborate in the exploitation attempt (*botnet*). Distributed attacks are one of the most difficult to handle by common firewalls, due to the diversity of request sources. As DRITAE does not take into account the source address of attackers, it has the same effectiveness when facing single-node or botnet attacks.

Although the primary source of diversification is the use of cross-development tools, it is also possible to create variants for native architectures using different compilation flags, and it may even be interesting to create some variants from former versions of the code.

Another possible improvement consists in adapting the policy for variant replacement when able to diagnose the particular kind of attack and the exploited vulnerability. In this way the most suitable architecture for facing each exploitation attempt could be selected from the variants pool.

Likewise, a simple improvement – decreasing the induced spatial overhead – could result in limiting launches to only two variants: the current and the next.

8.7 Conclusions and future work

Nowadays, memory errors rank among the most dangerous software errors despite vast research efforts made by academia and industry. Although existing protection mechanisms constitute a formidable barrier to the successful exploitation of memory errors by common hackers, these mechanisms do not constitute an impassable obstacle to more capable and resourceful opponents, such as black hat hackers and APT groups.

Thanks to protection mechanisms, most attacks are thwarted, albeit at the cost of causing a server crash. The administrator can configure the service to either stop at once, with related economic losses, or to keep it running, with an increasing likelihood of successful attacks. The work presented in this paper, relying on diversification, complements existing protection mechanisms with a detection and reaction approach that provides a third possibility, less drastic and dangerous, to hinder and even prevent the successful exploitation of memory errors while preserving service continuity.

Based on the fact that software faults leading to memory errors are highly dependent on the considered hardware architecture (processor), our technique uses processor diversification as effective protection against most kinds of attacks and accidental faults; in short, those attacks designed to target a specific processor will not succeed. This will force attackers to build meta-attacks for all the available target architectures, greatly hindering the possibility of a real attack. Even if such a meta-attack was available, applying a replacement policy in which the next variant is randomly selected prevents attackers from obtaining useful information to bypass existing mechanisms. The required processor diversification can be done in an efficient way thanks to current advances in processor emulation techniques.

Contrary to most automatic diversification techniques which customise the compiler or even the resulting executable binary, the use of cross-toolchains provides a simple and powerful solution for software diversification, with the benefit of using widely used and tested tools without any modification. In this way, and combined with underlying hardware diversification, existing software faults will manifest differently among variants, and it may not even manifest at all in some of them.

The feasibility and portability of this secure approach have been proven by deploying a HTTP Web server in two totally different scenarios: a personal computer and a smartphone. Results show that common attacks against existing protection mechanisms are effectively handled at the cost of degrading service performance due to processor emulator overheads. Service degradation, which is usually considered as a negative, undesired side-effect, also plays against attackers, as brute force attacks will be greatly delayed, thus giving administrators the time required to react while still providing the expected service.

The powerful capabilities provided by DRITAE open up a wide range of different possibilities for further research. For instance, i) a deep study of different hardware architectures from the perspective of how memory errors manifest is required to characterise their robustness against exploitation attempts, ii) precisely diagnosing the attack in a process could be of invaluable help in selecting the most suitable hardware architecture to hinder this attack or even prevent its success, and iii) the variant replacement policy could be tailored to fit the needs of particular services or scenarios.

Part IV

Conclusions

Chapter 9

Conclusions

9.1 General conclusions

This thesis has proposed several improvements to the designs of two of the most effective and widely used security protection techniques: SSP and ASLR, as well as a novel solution to the problem of automatic software diversification. All of the initial goals that were envisioned when the thesis work commenced have been fulfilled successfully.

In computer science, researchers focus mainly on new problems, new solutions and new paradigms, as well as how get one step ahead of the current state of the art. Furthermore, those solutions or techniques considered 'mature' do no receive enough attention. In this thesis, we took the risky path of exploring mature, well-tested and widely used solutions, and we were able to make significant advances in both theoretical and applied domains.

The dissemination of the results is not limited to academic papers only. Some of the obtained results have been transferred to the industry – to open source projects especially – in the form of contributed code, and a software patent has also been presented to the USPTO[1]. In addition, research efforts which resulted in discovering vulnerabilities have been acknowledged as CVEs and published in multiple security advisory repositories, while other results have been publicly published on two websites: the author's personal site and the group's website.

9.2 Contributions

9.2.1 Theoretical contributions

- A new dimension of entropy for the reference canary in the SSP technique has been proposed which eliminates the possibility of bypassing

[1]USPTO: United States Patent and Trademark Office.

the SSP technique using any kind of brute force attacks on forking servers. This new solution is called 'RenewSSP'.

- A detailed statistical study of the effectiveness of NX, SSP and ASLR has been presented. The study shows that the combined use of RenewSSP with the other two protection techniques (NX and ASLR) increases exponentially the level of protection against ASLR-targeted brute force attacks.
- The classic memory process memory model (fixed zones and growable areas) has been questioned, and a new memory layout model has been proposed, which in turn allowed us to redesign the existing ASLR technique.
- Multiple new dimensions of entropy have been proposed, which, jointly with the new layout model, resulted in a new ASLR protection technique design called 'ASLR-NG'.

  ASLR-NG is fully backward-compatible with all existing applications. The proposed ASLR algorithm is optimal in the sense that it provides maximum entropy for the memory layout that the MMU supports.
- A new form of automatic software diversification based on cross-compilers has been proposed (called DRITAE), which is simple to use, robust, cost-effective and easy to maintain.

9.2.2 Contributions to open source

- [PATCH] mm/x86: AMD Bulldozer ASLR fix. [**Linux 4.1**]
- [PATCH] x86, mm/ASLR: Fix stack randomization on 64-bit systems. [**Linux 4.0**]
- [PATCH] Preventing offset2lib attack. [**Linux 4.1**]
- [PATCH] BZ #15754: CVE-2013-4788: PTR_MANGLE does not initialise to a random value for the pointer guard when compiling static executables. [**GNU C library 2.19**]
- [PATCH] BZ #18928: Improper input validation of LD_POINTER_GUARD of set-user-ID and set-group-ID programs. [**GNU C library 2.22.90**]

9.2.3 Vulnerabilities discovered

Table 9.1 is a list of the vulnerabilities that are classified as 'security issues' by MITRE, in which case they receive a CVE number. We should also mention a few of the software errors found but not reported to the MITRE:

Id	**Product**	**Description**	**Cause**
CVE-2015-1593	Linux Kernel	Reduced stack entropy	Integer overflow
CVE-2015-1574	Email Android	Denial Of Service	Wrong data handling
CVE-2014-5439	sniffit	Root shell	Stack buffer overflow
CVE-2014-1226	s3dvt	Root shell (II)	Drop privileges failed
CVE-2013-6876	s3dvt	Root shell (I)	Drop privileges failed
CVE-2013-6825	DCMTK	Privilege escalation	Drop privileges failed
CVE-2013-4788	Eglibc	Bypass pointer guard	No pointer protection

Table 9.1: List of vulnerabilities evaluated as security issues: CVEs.

- The 'paxtest' tool, included in most Linux distributions, has two errors: entropy is incorrectly measured, and certain addresses are incorrectly obtained.
- In the Linux kernel, the data segment size resource (`RLIMIT_DATA`) set by the `setrlimit()` system call is incorrectly tested. This matter will be reported soon to the kernel list.

9.2.4 Patent

A software patent titled '*Method for Preventing Information Leaks in the Stack-Smashing Protector Technique*' was filed. It is currently under evaluation(patent pending: application patent number 14341118).

9.2.5 Academic Publications

- Héctor Marco-Gisbert and Ismael Ripoll. On the Effectiveness of Full-ASLR on 64-bit Linux '*In-depth security conference*' November 2014, (DeepSec 2014). PDF.
- Héctor Marco-Gisbert and Ismael Ripoll. On the effectiveness of nx, ssp, renewssp and aslr against stack buffer overflows. In '*13th International Symposium on Network Computing and Applications*', pages 145–152. IEEE, August 2014. ISBN 978-1-4799-5393-6.
- Héctor Marco-Gisbert and Ismael Ripoll. Preventing brute force attacks against stack canary protection on networking servers. In '*12th International Symposium on Network Computing and Applications*', pages 243–250, August 2013. doi: 10.1109/NCA.2013.12.
- Héctor Marco-Gisbert and Ismael Ripoll and David de Andrés and Juan Carlos Ruiz. '*Emerging trends in ICT security*', Elsevier Inc.

2013. Chapter 21, pp. 335–357. ISBN: 978-0-12-411474-6. doi: 10.1016/B978-0-12-411474-6.00021-9.

- Héctor Marco, Juan-Carlos Ruiz, David De Andrés and Ismael Ripoll. Preventing Memory Errors in Networked Vehicle Services Through Diversification '*Proceedings of Workshop CARS (2nd Workshop on Critical Automotive applications: Robustness & Safety)*' of the 32$^{\text{nd}}$ International Conference on Computer Safety, Reliability and Security, 2013.

9.2.6 Software tools and prototypes

During the work on the ASLR-NG, several tools were developed to analyse and validate the work.

ASLR Sampler: An application for collecting statistical information about all object mappings of the memory layout of processes.

ASLR Analyser: A powerful statistical tool employed to analyse data collected by the sampler application. It calculates typical statistical moments, makes several entropy estimations and calculates entropy correlations between all objects.

ASLR Simulator: A fast virtual memory allocator, used as a test bench during ASLR-NG design.

9.2.7 Honors & Awards

- The weakness of the ASLR design, jointly with a proof of concept exploitation code, was awarded by the **Packet Storm Security** bounty program and qualified as a **1-day** vulnerability. ASLR design had a weakness that allowed attackers to de-randomize (i.e. effectively bypass the ASLR) shared objects (libraries) by pivoting from application code. I named this weakness as Offset2lib. Packet-Storm-Advisory-2014-1204-1.

- The solution to the Offset2lib weakness included in the Linux kernel 4.0, was rewarded by the **Google** Patch Reward Program.

- The novel bit-slicing ASLR randomization form, presented in chapter 7 and applied in the Linux kernel 4.1, was rewarded by the **Google** Patch Reward Program. This new randomization form solves a security issue of the AMD Bulldozer processors.

- A security issue which reduces the ASLR entropy of the stack on 64 bit architectures was discovered and fixed in Linux kernel 4.0. This finding and its corresponding solution was rewarded by the **Google** Patch Reward Program.

9.3 Future work

This thesis represents the current status of most of the completed and consolidated results of an ongoing research activity. There are several currently active research issues, and just to mention a few:

- Contribute to the Apache project by fortifying servers using the RenewSSP technique.
- Include the ASLR-NG implementation on all processor architectures supported by Linux. The current implementation can be considered a 'proof of concept', but it needs a lot of work and 'iterations' with the Linux maintainers to make the code generic, robust and maintainable.
- Although ASLR-NG features are used transparently by most applications, with no need to modify or recompile the application code, others require the intervention of the library (intra-page on extended heap). The glibc may be modified to include these new features.
- Several helper tools have been developed during the analysis of the ASLR. These tools have proved to be very useful, but unfortunately they were developed to be used only internally and can not be distributed (no documentation, hard to configure, robust error handling, etc.). It would be interesting to publish this software.
- The analysis tools developed during the design of the ASLR-NG can be improved (made more user-friendly, robust and configurable) and published as a replacement for tools distributed by the PaX team.
- Regarding the DRITAE architecture, it would be possible to implement a more robust solution by running multiple variants simultaneously, feeding each one with the very same inputs and comparing the outputs for discrepancies. This software framework could be compared to the classic TRM (triple modular redundant) solution, widely used in hardware.

  A preliminary prototype of this solution has already been implemented (but not presented in this thesis, because it is still under development).

References

[1] Matt Blaze. Encryption technology and possible us policy responses. *US House of Representatives Committee on Government Oversight and Reform Information Technology Subcommittee*, April 2015.

[2] Common Weakness Enumeration (CWE), 2011. URL http://cwe.mitre.org.

[3] CWE/SANS. Top 25 most dangerous software errors, 2011. URL http://cwe.mitre.org/top25.

[4] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, pages 86–106, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33337-8. doi: 10.1007/978-3-642-33338-5_5. URL http://dx.doi.org/10.1007/978-3-642-33338-5_5.

[5] Karen Scarfone, Wayne Jansen, and Miles Tracy. Guide to general server security. *NIST Special Publication*, 800:123, 2008. URL http://books.google.es/books?id=XcFZLwEACAAJ.

[6] Shared Level - 1 instruction - cache performance on AMD family 15h CPUs, December 2011. URL http://developer.amd.com/wordpress/media/2012/10/SharedL1InstructionCacheonAMD15hCPU.pdf.

[7] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15 (1):2, 2012.

[8] The advanced return-into-lib(c) exploits, December 2001. URL http://phrack.org/issues/58/4.html.

[9] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[10] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended protection against stack smashing attacks without performance loss. In *In Proceedings of ACSAC*, 2006.

[11] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[12] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 56, 2002.

[13] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.

[14] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM. ISBN 1-58113-961-6. doi: 10.1145/1030083.1030124. URL `http://doi.acm.org/10.1145/1030083.1030124`.

[15] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, RAID'11, pages 121–141, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23643-3. doi: 10.1007/978-3-642-23644-0_7. URL `http://dx.doi.org/10.1007/978-3-642-23644-0_7`.

[16] Benjamin Erb. Concurrent programming for scalable web architectures. Diploma thesis, Institute of Distributed Systems, Ulm University, April 2012. URL `http://www.benjamin-erb.de/thesis`.

[17] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintongif, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th USENIX Security Symposium*, pages 63–78, Jan 1998.

[18] 'xorl'. Linux GLibC Stack Canary Values, 2010. URL `http://xorl.wordpress.com/2010/10/14/linux-glibc-stack-canary-values/`.

[19] H. Etoh. GCC extension for protecting applications from stack-smashing attacks (ProPolice), 2003. URL `http://www.trl.ibm.com/projects/security/ssp/`.

[20] Adam 'pi3' Zabrocki. Scraps of notes on remote stack overflow exploitation, November 2010. URL http://www.phrack.org/issues.html?issue=67&id=13#article.

[21] Yong-Joon Park and Gyungho Lee. Repairing return address stack for buffer overflow protection. In *Proceedings of the 1st conference on Computing frontiers*, CF '04, pages 335–342, New York, NY, USA, 2004. ACM. ISBN 1-58113-741-9. doi: 10.1145/977091.977139. URL http://doi.acm.org/10.1145/977091.977139.

[22] Jon Oberheide. A look at ASLR in Android Ice Cream Sandwich 4.0, Feb 2012. URL https://www.duosecurity.com/blog/a-look-at-aslr-in-android-ice-cream-sandwich-4-0.

[23] Hector Marco-Gisbert and Ismael Ripoll. On the effectiveness of nx, ssp, renewssp and aslr against stack buffer overflows. In *13th International Symposium on Network Computing and Applications*, pages 145–152. IEEE, August 2014. ISBN 978-1-4799-5393-6.

[24] Hector Marco-Gisbert and Ismael Ripoll. Preventing brute force attacks against stack canary protection on networking servers. In *12th International Symposium on Network Computing and Applications*, pages 243–250, August 2013. doi: 10.1109/NCA.2013.12.

[25] Hector Marco-Gisbert and Ismael Ripoll. On the effectiveness of full-aslr on 64-bit linux. In *In-depth security conference, DeepSec*, November 2014. URL http://cybersecurity.upv.es/attacks/offset2lib/offset2lib-paper.pdf.

[26] Yu Ding, Zhuo Peng, Yuanyuan Zhou, and Chao Zhang. Android low entropy demystified. In *IEEE International Conference on Communications, ICC 2014, Sydney, Australia, June 10-14, 2014*, pages 659–664, 2014. doi: 10.1109/ICC.2014.6883394. URL http://dx.doi.org/10.1109/ICC.2014.6883394.

[27] Poll: How often do you reboot?, 2014. URL http://www.androidcentral.com/poll-how-often-do-you-reboot.

[28] Damien Miller. Security Measures in OpenSSH, 2007. URL http://www.openbsd.org/papers/openssh-measures-asiabsdcon2007-slides.pdf.

[29] Adam Greenberg. SC Magazine: Trojanized Android apps steal authentication tokens, put accounts at risk, April 2014. URL www.scmagazine.com/trojanized-android-apps-steal-authentication-tokens-put-accounts-at-risk/article/342208/.

[30] Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. Address space randomization for mobile devices. In *Proceedings of the fourth ACM conference on Wireless network security*, WiSec '11, pages 127–138, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0692-8. doi: 10.1145/1998412.1998434. URL `http://doi.acm.org/10.1145/1998412.1998434`.

[31] Pax Team. PaX address space layout randomization (ASLR), 2003. URL `http://pax.grsecurity.net/docs/aslr.txt`.

[32] Linda Dailey Paulson. New chips stop buffer overflow attacks. *Computer*, 37(10):28–30, 2004.

[33] Mitre. CWE/SANS top 25 most dangerous software errors, 2011. URL `http://cwe.mitre.org/top25`.

[34] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 60–69, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3919-5. doi: 10.1109/ACSAC.2009.16. URL `http://dx.doi.org/10.1109/ACSAC.2009.16`.

[35] Jin Han, Debin Gao, and Robert H. Deng. On the effectiveness of software diversity: A systematic study on real-world vulnerabilities. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, pages 127–146, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02917-2. doi: 10.1007/978-3-642-02918-9_8. URL `http://dx.doi.org/10.1007/978-3-642-02918-9_8`.

[36] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 58, 2001.

[37] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012. ISSN 1094-9224. doi: 10.1145/2133375.2133377. URL `http://doi.acm.org/10.1145/2133375.2133377`.

[38] NIST. Vulnerability Summary for CVE-2010-3867, September 2011. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3867`.

[39] Eep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits.

In *In Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[40] Jakub Jelinek. Object size checking to prevent (some) buffer overflows (GCC FORTIFY), September 2004. URL http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html.

[41] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315313. URL http://doi.acm.org/10.1145/1315245.1315313.

[42] Christian W. Otterstad. Brute force bypassing of ASLR on 64-bit x86 GNU/Linux, November 2012. URL http://tapironline.no/last-ned/1081.

[43] Theo De Raadt. Exploit Mitigation Techniques (updated to include random malloc and mmap) at OpenCON 2005, 2005. URL http://www.openbsd.org/papers/ven05-deraadt/mgp00001.html.

[44] Kurt Miller. OpenBSD's Position Independent Executable (PIE) Implementation, 2008. URL http://www.openbsd.org/papers/nycbsdcon08-pie/mgp00001.html.

[45] Mark Russinovich. Inside the windows vista kernel: Part 3, 2007. URL http://technet.microsoft.com/en-us/magazine/2007.04.vistakernel.aspx.

[46] Ollie Whitehouse. An analysis of address space layout randomization on windows vista. Technical report, Symantec Advanced Threat Research, 2007. URL http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf.

[47] Clint Ruoho. Aslr: Leopard versus vista, 2008. URL http://www.laconicsecurity.com/aslr-leopard-versus-vista.html.

[48] Alexander Gabert Ned Ludd. Hardened/Introduction to Position Independent Code, 2013. URL http://wiki.gentoo.org/wiki/Hardened/Introduction_to_Position_Independent_Code.

[49] Mathias Payer and Thomas R. Gross. String oriented programming: When aslr is not enough. In *Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13,

pages 2:1–2:9, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1857-0. doi: 10.1145/2430553.2430555. URL `http://doi.acm.org/10.1145/2430553.2430555`.

[50] Steve McConnell. *Code Complete, Second Edition.* Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619670, 9780735619678.

[51] CVE-2013-2028. Nginx HTTP Server stack buffer overflow, July 2013. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2028`.

[52] CVE-2013-5019. Ultra Mini HTTPD stack buffer overflow, July 2013. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-5019`.

[53] CVE-2014-0063. PostgreSQL Multiple stack-based buffer overflows, February 2014. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0063`.

[54] CVE-2014-0065. PostgreSQL Multiple buffer overflows, February 2014. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0065`.

[55] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.*, 44(3):11:1–11:46, June 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187673. URL `http://doi.acm.org/10.1145/2187671.2187673`.

[56] Jesus Friginal, David de Andrés, Juan Carlos Ruiz, and Pedro J. Gil. Attack injection to support the evaluation of ad hoc networks. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*, pages 21–29. IEEE Computer Society, 2010. ISBN 978-0-7695-4250-8. doi: 10.1109/SRDS.2010.11. URL `http://dx.doi.org/10.1109/SRDS.2010.11`.

[57] J. Xu, Z. Kalbarczyk, and R.K. Iyer. Transparent runtime randomization for security. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 260–269, Oct 2003. doi: 10.1109/RELDIS.2003.1238076.

[58] Xun Zhan, Tao Zheng, and Shixiang Gao. Defending rop attacks using basic block level randomization. In *Software Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on*, pages 107–112, June 2014. doi: 10.1109/SERE-C.2014.28.

[59] Chongkyung Kil, Jinsuk Jim, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 339–348. IEEE, 2006.

[60] V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan. Preventing overflow attacks by memory randomization. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 339–347, Nov 2010. doi: 10.1109/ISSRE.2010.22.

[61] Jake Edge. Kernel address space layout randomization, October 2013. URL `https://lwn.net/Articles/569635`.

[62] Ulrich Drepper. Growable maps removal, August 2008. URL `https://lwn.net/Articles/294001`.

[63] Vincent Lefevre. Silent stack-heap collision under GNU/Linux, July 2014. URL `https://gcc.gnu.org/ml/gcc-help/2014-07/msg00076.html`.

[64] Chris Rohlf and Yan Ivnitskiy. Attacking clientside jit compilers. *Black Hat USA*, 2011.

[65] The Heartbleed Bug, April 2014. URL `http://heartbleed.com`.

[66] PaulR. Wilson, MarkS. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In HenryG. Baler, editor, *Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 1–116. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-60368-9. doi: 10.1007/3-540-60368-9_19. URL `http://dx.doi.org/10.1007/3-540-60368-9_19`.

[67] Hector Marco-Gisbert and Ismael Ripoll. AMD Bulldozer Linux ASLR weakness: Reducing entropy by 87.5%, March 2015. URL `http://hmarco.org/bugs/AMD-Bulldozer-linux-ASLR-weakness-reducing-mmaped-files-by-eight.html`.

[68] Hector Marco-Gisbert and Ismael Ripoll. CVE-2015-1593 - Linux ASLR integer overflow: Reducing stack entropy by four, January 2015. URL `http://hmarco.org/bugs/linux-ASLR-integer-overflow.html`.

[69] Hector Marco-Gisbert and Ismael Ripoll. Linux ASLR mmap weakness: Reducing entropy by half, January 2015. URL `http://hmarco.org/bugs/linux-ASLR-reducing-mmap-by-half.html`.

[70] ANNICK LESNE. Shannon entropy: a rigorous notion at the crossroads between probability, information theory, dynamical systems and statistical physics. *Mathematical Structures in Computer Science*, 24, 6 2014. ISSN 1469-8072. doi: 10.1017/S0960129512000783. URL `http://journals.cambridge.org/article_S0960129512000783`.

[71] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *In the Proceedings of the 15th International Symposium on Research in Attacks Intrusions and Defenses (RAID)*, September 2012.

[72] B. A. Wichmann. Requirements for programming languages in safety and security software standards. *Comput. Stand. Interfaces*, 14(5-6): 433–441, dec 1992. ISSN 0920-5489. doi: 10.1016/0920-5489(92)90009-3. URL `http://dx.doi.org/10.1016/0920-5489(92)90009-3`.

[73] Marc Brünink, Martin Süßkraut, and Christof Fetzer. Boundless memory allocations for memory safety and high availability. In *Proceedings of The 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2011)*, pages 13 –24, Los Alamitos, CA, USA, June 2011. IEEE Computer Society. doi: 10.1109/DSN.2011.5958203.

[74] Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*. USENIX Association, 2011. URL `http://dblp.uni-trier.de/db/conf/uss/uss2011.html#SnowKMP11`.

[75] Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz. Runtime defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing*, 8:588–601, 2011. ISSN 1545-5971. doi: http://doi.ieeecomputersociety.org/10.1109/TDSC.2011.18.

[76] Ryan Riley, Xuxian Jiang, and Dongyan Xu. An architectural approach to preventing code injection attacks. *IEEE Trans. Dependable Secur. Comput.*, 7(4):351–365, oct 2010. ISSN 1545-5971. doi: 10.1109/TDSC.2010.1. URL `http://dx.doi.org/10.1109/TDSC.2010.1`.

[77] David A. Patterson. A simple way to estimate the cost of downtime. In *Proceedings of the 16th USENIX conference on System administration*, LISA '02, pages 185–188, Berkeley, CA, USA, 2002. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1050517.1050538`.

[78] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: fault containment for shared-memory multiprocessors. *SIGOPS Oper. Syst. Rev.*, 29(5):12–25, dec 1995. ISSN 0163-5980. doi: 10.1145/224057.224059. URL `http://doi.acm.org/10.1145/224057.224059`.

[79] Jean-Claude Laprie, Christian Béounes, and Karama Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, jul 1990. ISSN 0018-9162. doi: 10.1109/2.56851. URL `http://dx.doi.org/10.1109/2.56851`.

[80] Paulo E. Verissimo, Nuno F. Neves, Christian Cachin, Jonathan Poritz, David Powell, Ives Deswarte, Robert Stroud, and Ian Welch. Intrusion tolerant middleware: The road to automatic security. *IEEE Security and Privacy*, 4(4):54–62, 2006.

[81] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassel. *The Shellcoder's handbook: Discovering and Exploiting Security Holes.* Wiley Publishing Inc., 2006. ISBN 978-0764544682.

[82] NIST. Vulnerability Summary for CVE-2011-2501, July 2011. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-2501`.

[83] David Brumley, Tzi cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Rich: Automatically protecting against integer-based vulnerabilities. In *In Symp. on Network and Distributed Systems Security*, 2007.

[84] NIST. Vulnerability Summary for CVE-2011-3026, September 2012. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3026`.

[85] NIST. Vulnerability Summary for CVE-2012-3547, November 2012. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3547`.

[86] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM. ISBN 1-58113-738-9. doi: 10.1145/948109.948146. URL `http://doi.acm.org/10.1145/948109.948146`.

[87] Benjamin Erb. Concurrent programming for scalable web architectures. Diploma thesis, Institute of Distributed Systems, Ulm University, April 2012. URL `http://www.benjamin-erb.de/thesis`.

[88] Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz. Runtime defense against code injection attacks using replicated execution. *IEEE Trans. Dependable Sec. Comput.*, 8 (4):588–601, 2011.

[89] CERT. Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability, June 2002. URL `http://www.cert.org/advisories/CA-2002-17.html`.

[90] Thorsten Glaser. busybox: integer overflow in expression on big endian. http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=635370, July 2011.

[91] Bruce Perens. Busybox. `http://www.busybox.net`, 1996.

[92] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX, 2005.

[93] CERT. Advisory CA-2002-08 Multiple Vulnerabilities in Oracle Servers, September 2002. URL `http://www.cert.org/advisories/CA-2002-08.html`.

[94] NIST. WFTPD Pro Server denial of service, May 2004. URL `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2004-0342`.

[95] Microsoft Security Bulletin. Buffer overrun in JPEG processing (GDI+) could allow code execution, September 2004. URL `http://technet.microsoft.com/en-us/security/bulletin/ms04-028`.

[96] Grsecurity kernel patches, 2013. URL `http://grsecurity.net/`. `http://grsecurity.net/`.