The “ptrace” Solution to Stack Integrity Attacks in GNU/Linux Systems

Erick Leon

A thesis submitted to the Department of

Computer Science in conformity with the requirements

for the degree of Master of Science

Bishop’s University

Sherbrooke, Quebec, Canada

May 2015

Abstract

Security in Information Technologies is an ever-changing field due to the fact that threats appear constantly with newer, more complex methods of attack as time goes on. Constant updates are therefore necessary on the systems which perform the operations, as well as the knowledge-pool of the personnel in charge of their security. With that in mind, it is also imperative to establish that while a team of human resources may be in charge of a system, they cannot realistically overview and monitor every single operation of it in order to detect anomalies or incidents that could jeopardize individual operations or in some cases, the entire system. As such, we argue that countermeasures against potential threats and attacks should be performed by the system itself while allowing it to remain useful. In addition, precise and concise notifications should be issued to the human resources in charge of such systems whenever threats and attacks are detected. That is the basis for our work: allow protection of a system while minimizing operational impact and issue proper notifications with precise data to the human resources.

The work outlined in this thesis addresses issues with cyber attack techniques that make use of certain logical address manipulations such as buffer overflows and code injection in GNU/Linux systems. The development of the solution explained in this document picks up where others left off, and so is mainly based on an earlier approach that was left at a proof-of-concept stage. We take this approach to a real-world scenario while attempting to remain true to the previous statements about the protection of systems and how feasible it can be.

Specifically, we develop a working, tested code that evaluates an executing process under GNU/Linux environments and whenever a call opcode has been detected, it stores and manages the corresponding stack pointer, which then gets validated once a ret opcode is found. This is performed during execution-time, which makes it a valuable tool to intercept buffer overflow attacks in real-time.

Acknowledgments

Por la patria, porque la patria es primero y por la raza.

My eternal gratitude to Dr. Stefan Bruda, for giving me a chance to pursue the M.Sc. degree under his outstanding guidance and patience... twice.

Thank you to Dr. Nelly Khouzam, for offering me the much-appreciated assistance when I needed it, and for being the Chair of the Committee for my thesis defense.

Thank you to professor Lin Jensen, for being an outstanding professor and for agreeing to evaluate my work; special thanks for the contribution that allowed for better performance.

Thank you to Dr. Ruben Gonzales-Rubio, for agreeing to evaluate my work and for the valuable insight into the matter.

Thank you to my parents, the masterminds behind all of my achievements.

Thank you to my family, for being the engine that keeps me moving forward.

And finally my eternal gratitude to Bishop's University, for being my home away from home and for allowing me to be part of this wonderful community.

Table of Contents

1 Introduction........1
2 Preliminaries........4
2.1 Assembly code........4
2.2 Memory addresses........4
2.3 Architectures........4
2.4 Registers........5
2.5 Opcodes........6
2.6 Stack........7
2.7 GNU/Linux........10
2.8 User space vs kernel space........11
2.9 Ptrace()........11
2.10 Buffer overflow and code injection........11
2.11 Python and the C programming languages........15
3 Previous work........16
3.1 Address Space Layout Randomization (ASLR)........16
3.2 Stack canaries........16
3.3 Bounds checking........17
3.4 Hardware-based protection........17
3.5 Dynamic Information Flow Tracking (DIFT)........17
3.6 Run-time/Executable monitoring........18
3.7 Never execute........18
3.8 Hardware-software hybrid methods........18
3.9 Array and pointer boundary checking........18
3.10 Counter-measures using kernel properties........18
4 Our Construction........20
4.1 Challenge........20
4.2 Procedure........20
4.3 Environment........21
4.4 Solution........21
4.4.1 Usage........21

4.4.2 Code......22
4.5 Faster......27
4.6 32-bit......30
5 Evaluation......31
5.1 Exploits......31
5.2 Benchmarking......32
5.3 Testing......32
5.3.1 OpenSUSE......34
5.3.2 Fedora......36
5.3.3 Ubuntu......39
5.4 Time......42
6 Conclusions......44
6.1 Limitations......45
6.1.1 Buffer optimization......47
6.2 Future work......48
7 Bibliography......50
8 Appendices......53
8.1 64fast.c......53
8.2 64full.c......55
8.3 32full.c......58
8.4 loop.c......60
8.5 vul.c......60
8.6 client.py......60
8.7 server.py......61

1 Introduction

As a species we have developed countless technologies and systems that allow us to simplify and automate tasks while constantly innovating and reinventing them as we see fit. In modern times we have witnessed the rise of drone and robotics technology for both military and civilian applications as well as in other areas of human activity, such as health care, marine biology and more [1]. The overall objective is simple: automate tasks that would be otherwise performed by humans, which allows for better resource and time management and in some cases, mitigates or transfers the risk to the automated system. Our work keeps up with the current trend of automated technologies. That is, we aim to automate a task that will protect a GNU/Linux operating system from a specific type of cyber attack. In order to accomplish this, some background is necessary.

The operating system GNU/Linux has been around for quite some time, it has been praised for having its code available to users, as well as for the vast number of devices on which it can operate. Linux is the kernel, while the GNU libraries are a set of files that work on top of the Linux kernel and allows it to perform tasks at a user-friendly level. In short, GNU/Linux is a proven and established operating system widely used by industries worldwide. As it would be expected from such a widely used system, malicious attempts at penetrating and modifying the system or operations under GNU/Linux are a common occurrence. It is there where Information Technology experts come into play by developing and constantly updating measures to both react to detected anomalies in the systems and prevent security incidents. Some of the viruses under GNU/Linux environments [2] date back to the mid-90's. In 1996 the virus Staog appeared and took advantage of a flaw in the kernel to attach itself to certain files. The 2000's saw the rise of more complex viruses with the Binom virus which expands the size of files, the Lupper worm which operates under Web servers and enables remote shell command execution, as well as many others. In the current decade we have seen the rise of even more integrated and complex viruses that take advantage of the habits of the contemporary user. Examples include the Koobface virus, which spreads through networking sites, and once having successfully infected a system attempts to obtain the credentials for File Transfer Protocol (FTP) or social networking sites, which are then used to spread it around. Evidently, as time goes on the complexity and scope of these viruses increase and the people in charge of protecting these systems are under an ever-changing, constant threat. It is here where we recall the current trend of automation technologies, and attempt to produce an applicable result to protect GNU/Linux systems from a specific type of cyber attack.

The type of attack we will detect automatically deals with stack pointers in GNU/Linux systems. A well-known example of this attack is a buffer overflow. A study that surveyed Common Vulnerability Scoring System (CVSS) data between 1988 and 2012 [33] showed that buffer overflows were the most often reported vulnerability, with 7,809 cases reported over the last 25 years, which amounts to roughly 14% of the 54,000 vulnerabilities that were assigned CVSS ratings. In further sections we will explore to a greater depth what it is and how it operates, but to sum it up buffer overflow attacks take advantage of flaws in the code of an executing process to overwrite values in memory. The outcome of an overflow usually leads to an error in the execution of the program, but if the attack is performed by a malicious entity the overwriting process could lead to the execution of code which is alien to the actual process being executed. Another type of attack is code injection, which uses flaws in either the code of the process being executed, or takes advantage of other methods like memory manipulation to overwrite code into the current executing process which leads once again to malicious code being executed. The common denominator for these attacks is the use of stack pointers. Therefore, we establish the following objectives for our work:

1) Develop a countermeasure against stack pointer manipulation in the GNU/Linux operating systems.

2) Automate the detection process during execution-time.

3) Notify the system administrator in real-time.

4) Minimize the consumption of system resources.

With the objectives laid out, we then develop our solution using the ptrace tool as our main framework. With it, we are capable of performing a number of tasks that will allow us to execute a process, halt it, review it, perform a desired task, and proceed to either the very next instruction or until we find a specified instruction or system call. This is all performed during execution and in real time, which in turn allows us to react appropriately. For our case, should an anomaly be detected we stop the process and issue a notification through the network to the system administrator with precise information that should the need arise can be potentially valuable during a forensic examination. This execution-time approach offers an alternate perspective to already-implemented approaches such as Address Space Layout Randomization (ASLR) and stack canaries, which rely on preventing issues before they occur. By contrast, our approach relies on reacting to anomalies whenever they happen.

In order to understand these mechanics to a greater depth, we will next proceed to overview the necessary concepts and background (Chapter 2). The previous work related to our pursuit is reviewed in Chapter 3. Our solution is then presented in Chapter 4 and subsequently evaluated in Chapter 5. Comments on the advantages and limitations of the proposed solution are included in Chapter 6.

2 Preliminaries

2.1 Assembly code

According to Techopedia [3] “An assembly language is a low-level programming language for microprocessors and other programmable devices. It is not just a single language, but rather a group of languages. Assembly language implements a symbolic representation of the machine code needed to program a given CPU architecture.”

These languages are usually quite complex and lack high-level conveniences such as variables and functions. They also change according to the processor they run on. Due to this being the most basic programming language available for a given processor, it makes it especially useful when dealing directly with the Central Processing Unit (CPU) and its operations.

2.2 Memory addresses

Following once more the simple definitions given by Techopedia [4], “A memory address is a unique identifier used by a device or CPU for data tracking. This binary address is defined by an ordered and finite sequence allowing the CPU to track the location of each memory byte.”

In short, this is the nomenclature used to reference the memory spaces available in the system.

2.3 Architectures

Architectures are a set of instructions developed for computer processors [5]. The architectures that we will work with are the 32-bit architecture for Intel processors (usually defined as x86) and the

64-bit architecture for the same family of processors (usually named x64). It should be noted however that other architectures also exist.

The difference between 32-bit and 64-bit architectures are the size of the data units, which is 32 bits and 64 bits, respectively. Operating systems like Windows 95, 98 and XP were common on computers with 32-bit processors. Operating systems like Windows Vista, Windows 7 and Windows 8 come in 64-bit versions. It is important to note that while software developed for 64-bit systems cannot be executed in 32-bit machines, 32-bit software can potentially be executed in 64-bit machines if certain necessary conditions are met; however, one cannot not run 16-bit legacy programs on 64-bit machines [6]. Another difference is the maximum amount of Random Access Memory (RAM) that is supported. For 32-bit computers the maximum amount is three to four GB of memory, while for 64-bit computers, they can support amounts well over four GB. This also translates into faster processing for 64-bit architectures. There are many more differences, especially at the technical level; however this information should suffice as a general overview for the purposes of our work. It is worth mentioning that GNU/Linux distributions are usually available for both architectures.

2.4 Registers

The operations undertaken by a processor involve processing data. Such data is stored in the RAM, which is orders of magnitude slower than the processor. Registers are therefore built into the processor chip. Their purpose is to store data elements without accessing the main memory; this improves processing time considerably.

Each architecture has its own set of registers. It is also important to note that there are various types of registers, for example: segment registers, control registers, index registers, pointer registers and more. For the purpose of our work, the registers that we will be dealing with are the pointer registers. Pointer registers can be classified in three categories [7]:

1) Instruction pointer: Stores the address of the next instruction to be executed.

2) Stack pointer: Stores the current position within the program stack.

3) Base pointer: References the parameter values passed to a subroutine.

Data registers change from architecture to architecture and are used for arithmetic, logical and other operations. We will only work with 32-bit and 64-bit architectures, so we provide this brief technical overview only for these architectures.

For 32-bit architectures, the data registers are EAX, EBX, ECX, and EDX. These can be used in three ways: we can use them as complete 32-bit data registers, the lower halves can be used as four 16-bit data registers AX, BX, CX and DX, or the lower and higher halves of these 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH and DL.

For the purpose of our work, 64-bit machines operate in effectively the same way. Nonetheless some changes in the use of registers do happen, meaning that while a register for a 32-bit machine is EAX, the equivalent for a 64-bit machine would be RAX. A stack pointer for a 32-bit machine would be ESP, while for a 64-bit machine it will be RSP. To summarize the difference with an extremely rough definition: "E" is for 32-bit and "R" for 64-bit. However it must be said that there are several other differences; given the complexity of the topic, this rough definition was given in order to over-simplify a complex definition into something that applies to our work, given the established constraints and scope.

2.5 Opcodes

Opcodes are values that represent procedures in the execution of a process. For the purposes of our work, we will only deal with the opcodes for call and ret:

Instruction	Opcode
Call	E8
	FF
	9A
	REX.W + FF
Ret	C3
	CB
	C2
	CA

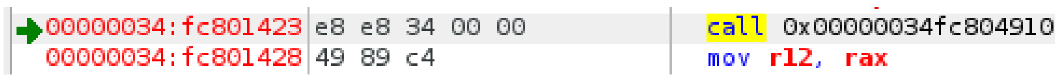
Table 1: Opcodes

These vary from architecture to architecture, but as previously mentioned we will only deal with the 32-bit and 64-bit architectures and therefore we refer to the official Intel manuals [8], which contain the list of all possible opcodes as well as their explanation. The purpose of the opcodes mentioned above will be exemplified later (in the next section) due to their role in the execution of a process. We can validate that these opcodes are contained within the long strings of hexadecimal values which represent the instructions to be executed.

2.6 Stack

The stack is defined as a push-down or Last In First Out (LIFO) list that contains data [9]. For the purpose of our work we will only consider the stack where the hexadecimal values of the memory addresses associated with calls and rets are stored, so we refer henceforth to this stack simply as “the stack”. We can visualize the behavior of the stack by following the execution of a process at an assembly code-level, as follows (in this example, the processor is little-endian):

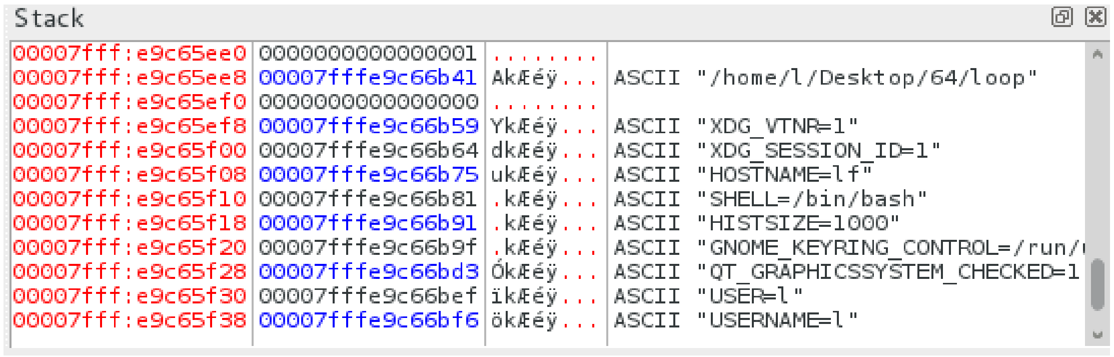
We begin the trace of the execution flow from a call:



```
00000034:fc801423 e8 e8 34 00 00    call 0x00000034fc804910
00000034:fc801428 49 89 c4          mov r12, rax
```

Illustration 1: Call execution

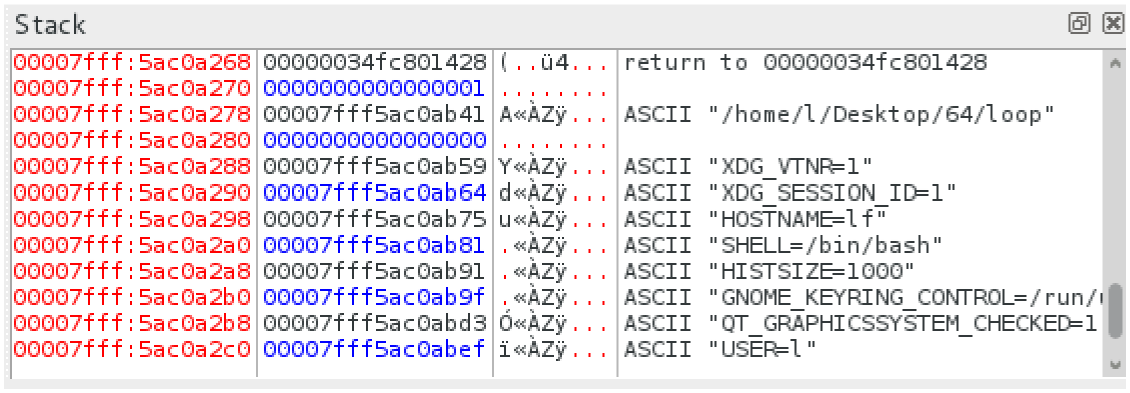
At this point (that is, before the execution of the call) we will check the stack:



```
Stack
00007fff:e9c65ee0 0000000000000001 ........
00007fff:e9c65ee8 00007fffe9c66b41 AkÆéÿ... ASCII "/home/l/Desktop/64/loop"
00007fff:e9c65ef0 0000000000000000 ........
00007fff:e9c65ef8 00007fffe9c66b59 YkÆéÿ... ASCII "XDG_VTNR=1"
00007fff:e9c65f00 00007fffe9c66b64 dkÆéÿ... ASCII "XDG_SESSION_ID=1"
00007fff:e9c65f08 00007fffe9c66b75 ukÆéÿ... ASCII "HOSTNAME=lf"
00007fff:e9c65f10 00007fffe9c66b81 .kÆéÿ... ASCII "SHELL=/bin/bash"
00007fff:e9c65f18 00007fffe9c66b91 .kÆéÿ... ASCII "HISTSIZE=1000"
00007fff:e9c65f20 00007fffe9c66b9f .kÆéÿ... ASCII "GNOME_KEYRING_CONTROL=/run/
00007fff:e9c65f28 00007fffe9c66bd3 ÓkÆéÿ... ASCII "QT_GRAPHICSSYSTEM_CHECKED=1
00007fff:e9c65f30 00007fffe9c66bef ïkÆéÿ... ASCII "USER=l"
00007fff:e9c65f38 00007fffe9c66bf6 ökÆéÿ... ASCII "USERNAME=l"
```

Illustration 2: Stack viewer

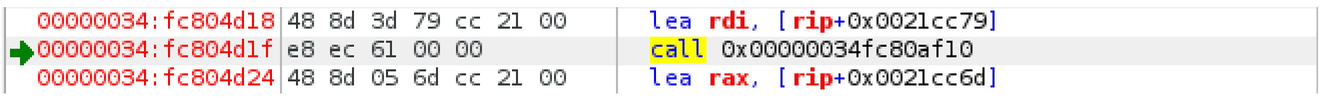
As we can see the stack has no stored value at the beginning of our execution. We will now execute the call and then we will check the stack again:



```
Stack
00007fff:5ac0a268 00000034fc801428 (..ü4... return to 00000034fc801428
00007fff:5ac0a270 0000000000000001 ........
00007fff:5ac0a278 00007fff5ac0ab41 A«ÀZÿ... ASCII "/home/l/Desktop/64/loop"
00007fff:5ac0a280 0000000000000000 ........
00007fff:5ac0a288 00007fff5ac0ab59 Y«ÀZÿ... ASCII "XDG_VTNR=1"
00007fff:5ac0a290 00007fff5ac0ab64 d«ÀZÿ... ASCII "XDG_SESSION_ID=1"
00007fff:5ac0a298 00007fff5ac0ab75 u«ÀZÿ... ASCII "HOSTNAME=lf"
00007fff:5ac0a2a0 00007fff5ac0ab81 .«ÀZÿ... ASCII "SHELL=/bin/bash"
00007fff:5ac0a2a8 00007fff5ac0ab91 .«ÀZÿ... ASCII "HISTSIZE=1000"
00007fff:5ac0a2b0 00007fff5ac0ab9f .«ÀZÿ... ASCII "GNOME_KEYRING_CONTROL=/run/
00007fff:5ac0a2b8 00007fff5ac0abd3 Ó«ÀZÿ... ASCII "QT_GRAPHICSSYSTEM_CHECKED=1
00007fff:5ac0a2c0 00007fff5ac0abef ï«ÀZÿ... ASCII "USER=l"
```

Illustration 3: Stack viewer

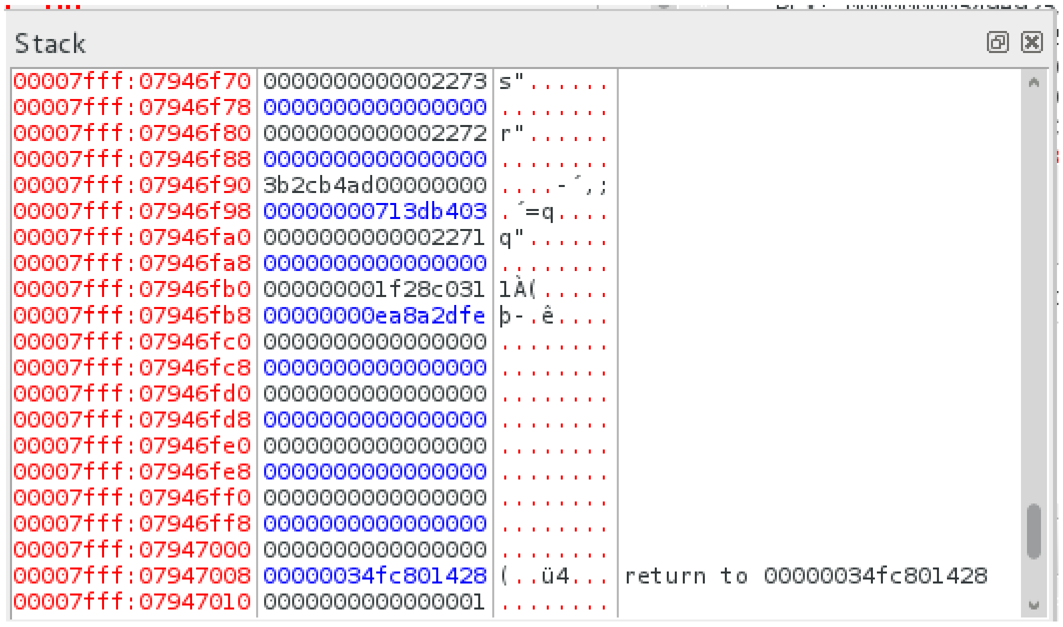
We now have a value stored on the stack. Indeed, whenever a call is issued in the execution of a process, a return value will be stored on the stack in order to allow the process to come back (return) to it. This is the behavior of the call/ret pair. We will now continue the execution process until another call is found:



```
 00000034:fc804d18 48 8d 3d 79 cc 21 00    lea rdi, [rip+0x0021cc79]
 00000034:fc804d1f e8 ec 61 00 00          call 0x00000034fc80af10
 00000034:fc804d24 48 8d 05 6d cc 21 00    lea rax, [rip+0x0021cc6d]
```

Illustration 4: Call execution

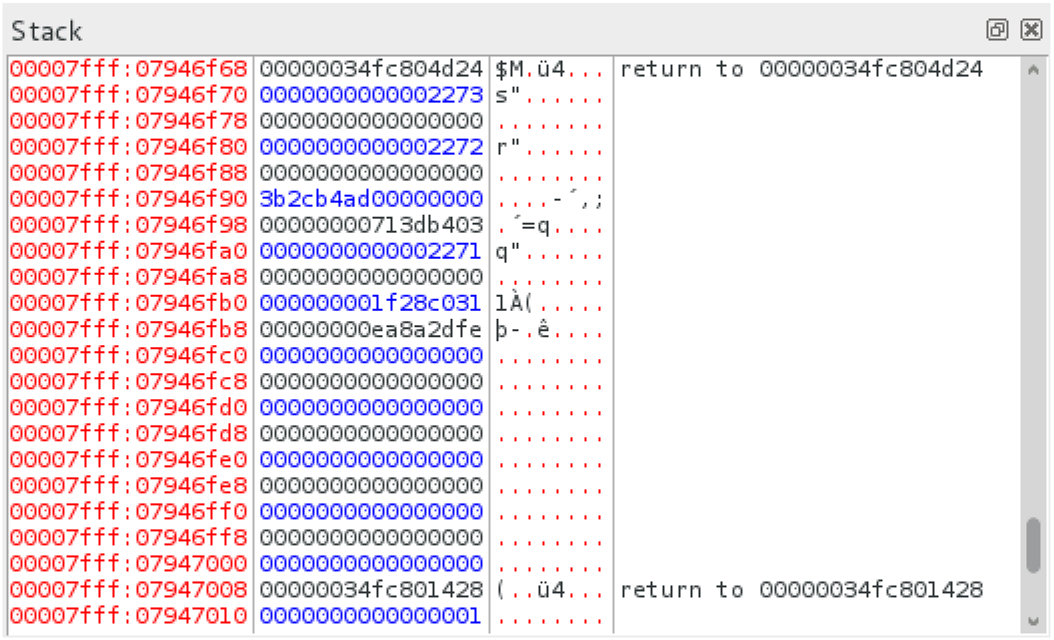
We are expecting the stack to keep storing data as a push-down list. Let us visualize the stack at this stage:



```
Stack
00007fff:07946f70 0000000000002273 s"......
00007fff:07946f78 0000000000000000 ........
00007fff:07946f80 0000000000002272 r"......
00007fff:07946f88 0000000000000000 ........
00007fff:07946f90 3b2cb4ad00000000 ....-´,;
00007fff:07946f98 00000000713db403 .´=q....
00007fff:07946fa0 0000000000002271 q"......
00007fff:07946fa8 0000000000000000 ........
00007fff:07946fb0 000000001f28c031 1À(.....
00007fff:07946fb8 00000000ea8a2dfe þ-.ê....
00007fff:07946fc0 0000000000000000 ........
00007fff:07946fc8 0000000000000000 ........
00007fff:07946fd0 0000000000000000 ........
00007fff:07946fd8 0000000000000000 ........
00007fff:07946fe0 0000000000000000 ........
00007fff:07946fe8 0000000000000000 ........
00007fff:07946ff0 0000000000000000 ........
00007fff:07946ff8 0000000000000000 ........
00007fff:07947000 0000000000000000 ........
00007fff:07947008 00000034fc801428 (..ü4... return to 00000034fc801428
00007fff:07947010 0000000000000001 ........
```

Illustration 5: Stack viewer

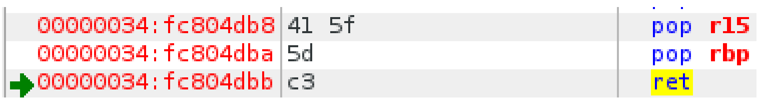
Evidently, some values have been pushed onto the stack, but since we are following calls, we now expect to have another ret value stored into the top of the stack when we execute the call. Indeed, checking the stack at this point in the execution (after the call is executed) confirms this:



```
Stack
00007fff:07946f68 00000034fc804d24 $M.ü4... return to 00000034fc804d24
00007fff:07946f70 0000000000002273 s"......
00007fff:07946f78 0000000000000000 ........
00007fff:07946f80 0000000000002272 r"......
00007fff:07946f88 0000000000000000 ........
00007fff:07946f90 3b2cb4ad00000000 ....-´,;
00007fff:07946f98 00000000713db403 .´=q....
00007fff:07946fa0 0000000000002271 q"......
00007fff:07946fa8 0000000000000000 ........
00007fff:07946fb0 00000001f28c031 1À(.....
00007fff:07946fb8 00000000ea8a2dfe þ-.ê....
00007fff:07946fc0 0000000000000000 ........
00007fff:07946fc8 0000000000000000 ........
00007fff:07946fd0 0000000000000000 ........
00007fff:07946fd8 0000000000000000 ........
00007fff:07946fe0 0000000000000000 ........
00007fff:07946fe8 0000000000000000 ........
00007fff:07946ff0 0000000000000000 ........
00007fff:07946ff8 0000000000000000 ........
00007fff:07947000 0000000000000000 ........
00007fff:07947008 00000034fc801428 (..ü4... return to 00000034fc801428
00007fff:07947010 0000000000000001 ........
```

Illustration 6: Stack viewer

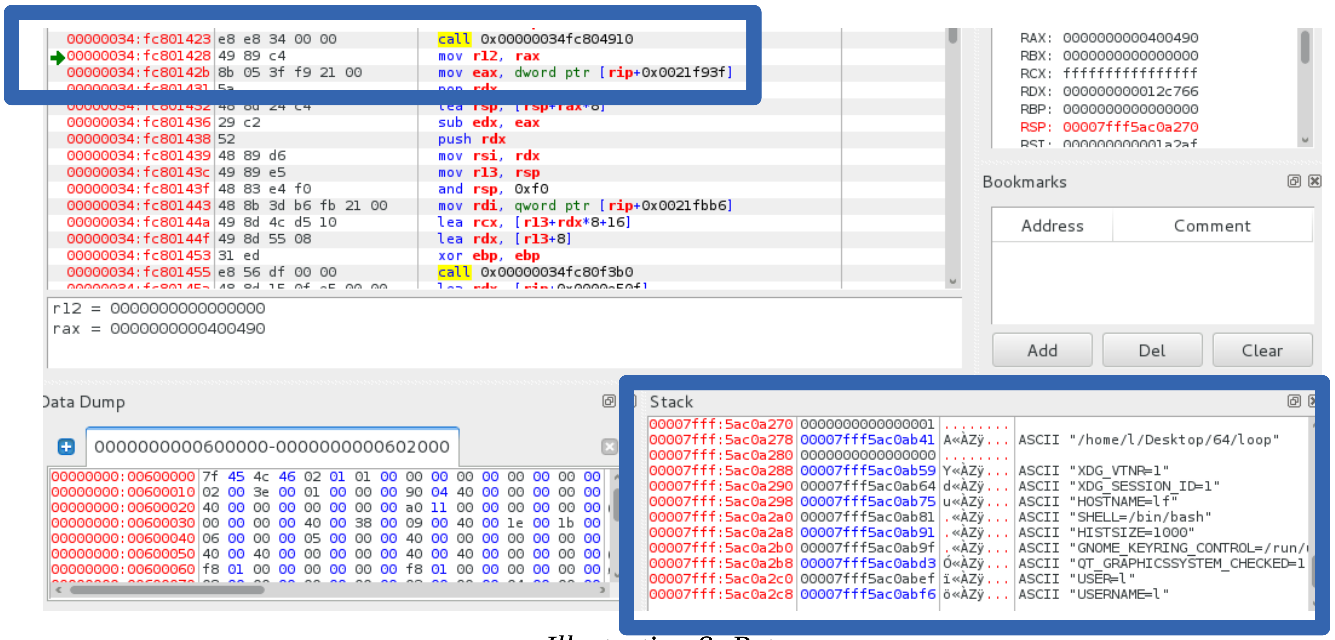
We continue the execution flow until we find a ret to the original call from Illustration 1:



```
00000034:fc804db8 41 5f    pop r15
00000034:fc804dba 5d       pop rbp
00000034:fc804dbb c3       ret
```

Illustration 7: Return

When we execute it, we should return to the address 34fc801428:



```
00000034:fc801423 e8 e8 34 00 00        call 0x00000034fc804910
00000034:fc801428 49 89 c4              mov r12, rax
00000034:fc80142b 8b 05 3f f9 21 00     mov eax, dword ptr [rip+0x0021f93f]
00000034:fc801436 29 c2                 sub edx, eax
00000034:fc801438 52                    push rdx
00000034:fc801439 48 89 d6              mov rsi, rdx
00000034:fc80143c 49 89 e5              mov r13, rsp
00000034:fc80143f 48 83 e4 f0           and rsp, 0xf0
00000034:fc801443 48 8b 3d b6 fb 21 00  mov rdi, qword ptr [rip+0x0021fbb6]
00000034:fc80144a 49 8d 4c d5 10        lea rcx, [r13+rdx*8+16]
00000034:fc80144f 49 8d 55 08           lea rdx, [r13+8]
00000034:fc801453 31 ed                 xor ebp, ebp
00000034:fc801455 e8 56 df 00 00        call 0x00000034fc80f3b0

r12 = 0000000000000000
rax = 0000000000400490
```

```
RAX: 0000000000400490
RBX: 0000000000000000
RCX: ffffffffffffffff
RDX: 000000000012c766
RBP: 0000000000000000
RSP: 00007fff5ac0a270
```

```
00000000:00600000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
00000000:00600010 02 00 3e 00 01 00 00 00 90 04 40 00 00 00 00 00
00000000:00600020 40 00 00 00 00 00 00 00 a0 11 00 00 00 00 00 00
00000000:00600030 00 00 00 00 40 00 38 00 09 00 40 00 1e 00 1b 00
00000000:00600040 06 00 00 00 05 00 00 00 40 00 00 00 00 00 00 00
00000000:00600050 40 00 40 00 00 00 00 00 40 00 40 00 00 00 00 00
00000000:00600060 f8 01 00 00 00 00 00 00 f8 01 00 00 00 00 00 00
```

```
Stack
00007fff:5ac0a270 0000000000000001 ........
00007fff:5ac0a278 00007fff5ac0ab41 A«ÀZÿ... ASCII "/home/l/Desktop/64/loop"
00007fff:5ac0a280 0000000000000000 ........
00007fff:5ac0a288 00007fff5ac0ab59 Y«ÀZÿ... ASCII "XDG_VTNR=1"
00007fff:5ac0a290 00007fff5ac0ab64 d«ÀZÿ... ASCII "XDG_SESSION_ID=1"
00007fff:5ac0a298 00007fff5ac0ab75 u«ÀZÿ... ASCII "HOSTNAME=lf"
00007fff:5ac0a2a0 00007fff5ac0ab81 .«ÀZÿ... ASCII "SHELL=/bin/bash"
00007fff:5ac0a2a8 00007fff5ac0ab91 .«ÀZÿ... ASCII "HISTSIZE=1000"
00007fff:5ac0a2b0 00007fff5ac0ab9f .«ÀZÿ... ASCII "GNOME_KEYRING_CONTROL=/run/
00007fff:5ac0a2b8 00007fff5ac0abd3 Ó«ÀZÿ... ASCII "QT_GRAPHICSSYSTEM_CHECKED=1
00007fff:5ac0a2c0 00007fff5ac0abef ï«ÀZÿ... ASCII "USER=l"
00007fff:5ac0a2c8 00007fff5ac0abf6 ö«ÀZÿ... ASCII "USERNAME=l"
```

Illustration 8: Return

Indeed, we are now back to the corresponding address as expected and the value in the stack was pushed out.

2.7 GNU/Linux

Linux is a kernel, meaning that it is a program that manages the resources of a system (in other words, the core). GNU can be defined in many ways but to to give the briefest description it is a series of libraries that operate on top of the Linux kernel, which in turn allows this kernel to perform many tasks that would otherwise not be possible [10]. On top of that, we get distributions, which are essentially versions of GNU/Linux that are packaged with useful software like multimedia players and more.

The combination of both is commonly referred to just as Linux. However, they are as we already mentioned different. GNU/Linux comes in many versions which are called distributions. A relatively popular distribution is Ubuntu so it might be easier to recognize GNU/Linux by this name for the purpose of this section. However, since one of our objectives is to consider GNU/Linux as a whole, we will focus on the top three most popular distributions, which are Ubuntu, Fedora and OpenSUSE [11]. While an argument could be made that more distributions such as Linux Mint, Debian, RedHat, CentOS, Arch, etc. are also popular, we had to limit the scope of our work in order to avoid unnecessary consumption of resources and time. It is also worth mentioning that some of these distributions are themselves based off other distributions, for example Fedora and CentOS are based off RedHat, also Ubuntu and Linux Mint are based off Debian and so on and so forth.

While an argument could be made towards the use of base distributions like RedHat, we argue that they are targeted towards individuals or even companies with specialized needs and operations; this potential target group would have contracted the professional services of technical support from the seller, and in some cases acquired certifications for their human resources. Therefore, the possibility of such an advantage between users of these distributions goes against our objective of showing our solution working “universally” across GNU/Linux platforms and their users. In short, the decision to use these three popular yet “watered-down” or “simplified” distributions was reached by a combination of popularity, availability and personal preference, thus any argument for or against our choice is potentially valid but tangent to the substance of our work.

2.8 User space vs kernel space

Kernel space refers to the environment that interacts directly with the running kernel, which is the core of the operating system [12]. The kernel has full access to all memory and other hardware of the system and so kernel space is an extremely sensitive environment where only the most trusted and well-tested code runs. Should a problem arise in kernel space, the entire system could be jeopardized. On the other hand, user space refers to the environment in which most software runs, which operates under restricted access to resources and functions under a form of sandboxing. Should problems arise in user space, the system would not necessarily be in jeopardy. Naturally, this segregation between these environments was created to prevent problems and mitigate risks.

2.9 Ptrace()

From the manual page of ptrace [13]: “The ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change in the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.”

Regarding its application in our work, ptrace is a built-in tool in GNU/Linux operating systems and will let us intercept certain resources in a process for analysis. This will become clearer a bit later when our solution is explained.

2.10 Buffer overflow and code injection

A buffer is an area of memory used to store data [14]. Buffer overflow happens when a program attempts to put more data in a buffer than it can store, or when a program attempts to put data in a memory area past a buffer. Possible results of a buffer overflow can vary from data corruption, to crashing of the program, to malicious code execution. The latter result is the main concern of and motivation for our work.

As mentioned earlier buffer overflow is a well-known and well-understood software vulnerability and yet it is still a common occurrence. The problem resides in the actual development of the software and the fact that a buffer overflow can occur in a variety of ways. In practice, these vulnerabilities are not easily detected and when discovered they can be quite difficult to exploit. Regardless, this is still a significant issue in Information Technology security given the dangerous effect of malicious exploitation of buffer overflows.

The classic version of buffer overflow exploitation consists of an attacker sending data which is stored by the receiving program in an undersized stack buffer. The resulting sequence or actions leads to having information from the call stack overwritten, including the function's return pointer. The value thus set by the overflowing data becomes a return pointer that leads the execution flow to an arbitrary memory location, that could be malicious code contained in the attacker's data, essentially handing over control to the malicious code.

There are many forms of buffer overflow such as Heap buffer overflow, Off-by-one Error, Format string attacks and others [15], but for the purposes of our work only those attacks that deal with the corruption of the stack pointers will be taken into account.

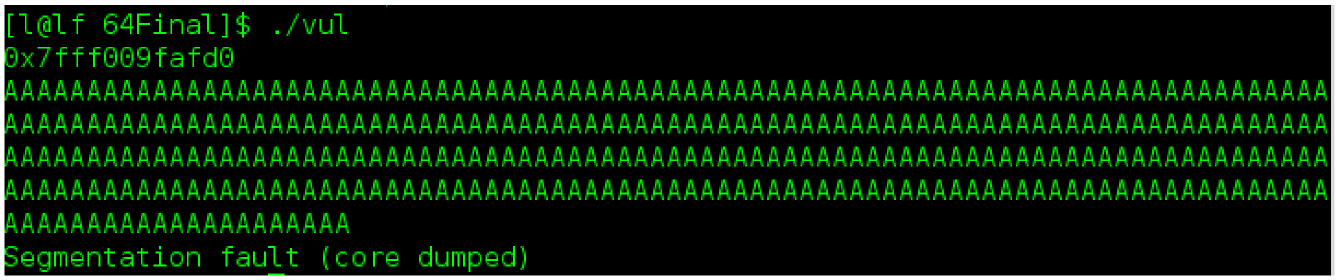
We will now review a scenario where a buffer overflow occurs and is then exploited. Consider the following C program:

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int main(int argc, char **argv) {

   char buffer[256];

   return 0;

}
```

In this scenario we have a program written in C that creates a buffer with room for 256 characters. If we introduce more characters that it can hold, it should overflow. We do that by adding the following piece of code to the above program:

```
strcpy(buffer,
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
");

printf("%s\n", buffer);
```

There are roughly 300 or more As in the string being copied in the buffer. If we compile and execute the program, we should see an overflow message:



```
[l@lf 64Final]$ ./vul
0x7fff009fafd0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

Illustration 9: Buffer overflow

Indeed, the Segmentation fault (core dumped) message is caused by a segmentation violation of the program which is in turn caused by the overflow.

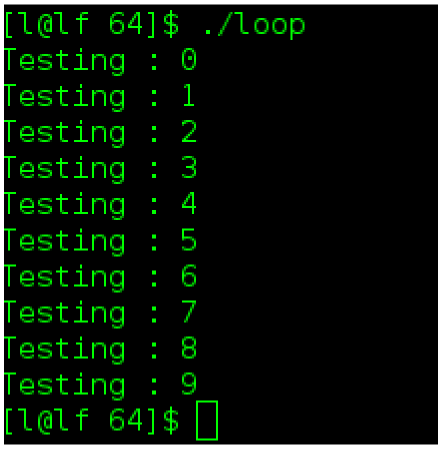
Code injection [16] is a generic term for certain types of attacks that consist of injecting code into a running program or process. Generally these attacks exploit vulnerabilities associated to poor handling of untrusted data. For the purposes of our work, only the types of code injection that deal with stack pointer manipulation will be considered.

We can exemplify this concept by injecting code into a process and then visualizing the behavior, as follows:

```
#include <stdio.h>

#include <stdlib.h>

int main() {

  int i;

  for(i = 0;i < 10; ++i) {

    printf("Testing : %d\n", i);

    sleep(2);

  }

  return 0;
```

}

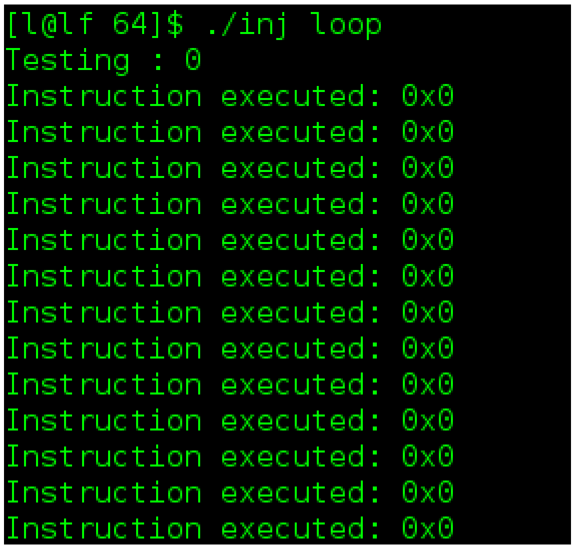
Here we have a simple program that will print to the terminal numbers from one to nine while pausing briefly in-between. When we compile and run it we see the following:



```
[l@lf 64]$ ./loop
Testing : 0
Testing : 1
Testing : 2
Testing : 3
Testing : 4
Testing : 5
Testing : 6
Testing : 7
Testing : 8
Testing : 9
[l@lf 64]$ 
```

Illustration 10: Loop

We will now inject code using ptrace. This will be explained in later sections; however the resulting visualization will be:



```
[l@lf 64]$ ./inj loop
Testing : 0
Instruction executed: 0x0
Instruction executed: 0x0
Instruction executed: 0x0
Instruction executed: 0x0
Instruction executed: 0x0
Instruction executed: 0x0
Instruction executed: 0x0
Instruction executed: 0x0
Instruction executed: 0x0
Instruction executed: 0x0
Instruction executed: 0x0
Instruction executed: 0x0
Instruction executed: 0x0
```

Illustration 11: Loop with code injection

This is caused by the injected code, which sends the execution flow to a random memory address after the first iteration, and tells it to execute the instruction stored there. However since there are no instructions to be executed at that address, we force the control flow to remain there. In a malicious scenario, an attacker would tell the control flow to go to an address where malicious code resides, so that this malicious code will be executed by the process.

2.11 Python and the C programming languages

The Python programming language is considered an easy language for beginners to use [17]. One of the main benefits of using Python is its overall simplicity and especially its powerful functions and other language constructs. Indeed, the number of lines of code is normally reduced in Python compared to other programming languages such as Java. A good case in point is the code necessary to handle network sockets (which is exactly what will be used in our solution).

The C programming language [18] is a widely-known language that has been tried and tested over decades. The main reason for our use of C in our solution is due to the fact that we use ptrace which can be easily accessed through this programming language. Also it is worth mentioning that a substantial amount of software, from the kernel to even the Python interpreter, is coded in C.

3 Previous work

In this section we address different solutions which offer new perspectives and approaches in the attempt of solve the problem of buffer overflow.

3.1 Address Space Layout Randomization (ASLR)

ASLR [19] operates under the idea that exploits usually rely on values such as addresses which are known to contain certain resources or pointers to be static and placed in known locations on the stack and elsewhere. ASLR introduces randomness into the virtual memory layout of a process in order to change the binary mapping of stack memory regions and dynamic library linking before the process executes in order to nullify any attacks that operate under the assumption of static values.

3.2 Stack canaries

Stack canaries are values that are inserted into specific, known locations in memory, usually near return addresses, in order to detect buffer overflows due to the fact that when one happens the first data to be corrupted will usually be the canary. They were originally implemented by Immunix Inc/WireX in the StackGuard GCC patches. Examples of these canaries are [20], [21], [22]:

1) Null Canaries: The value is set to 0x00000000 due to the fact that usually, string functions terminate on a null value and thus they would not be able to modify the return address.

2) Terminator Canaries: The value is set to Null, CR, LF and/or 0xFF. This leads to the attacker having to write a null character before modifying the return address. In essence this accounts for functions that do not terminate on null values, such as gets().

3) Random Canaries: Essentially each canary is a 32-bit random value generated by the system.

4) Random XOR Canaries: Performs an XOR operation on the canary value with the control data.

3.3 Bounds checking

Bounds checking [23] operates by keeping track of the address boundaries for objects, buffers and arrays, and constantly checking the load and store operations that access those resources. All the loads and stores are validated for whether they access a location inside the resource boundaries. The boundaries are usually represented by a lower and upper address; should the accessed memory address be outside of these boundaries, the system may either issue an exception or circumvent the error.

The application of bounds checking to buffer overflow attacks operates on the premise that a buffer overflow attack would point to a return address outside of the bounds and thus it would be detected.

3.4 Hardware-based protection

SecureBit2 [24] is an approach against buffer overflow attacks that operates on top of Secure Bit. Specifically, Secure Bit introduces a hardware bit to protect the integrity of addresses, while SecureBit2 works to protect Secure Bit and therefore successfully detect and prevent all buffer overflow attacks. This solution requires hardware support. The authors only provide proof-of-concept evidence of effectiveness and feasibility.

3.5 Dynamic Information Flow Tracking (DIFT)

The idea behind DIFT [25] is to tag untrusted data in order to track it through the system. A hierarchical model is also in place so that new data that is derived from untrusted data can be generated. This model has both software and hardware implementations. Current DIFT systems operate based on bound check recognition in order to detect buffer overflows. However this leads to many false positives and also some false negatives.

3.6 Run-time/Executable monitoring

Chaperon [26] and Valgrind are commercial tools that intercept malloc and free calls directly from a binary executable. A problem with Chaperon is that some overflows are reported incorrectly because the monitoring of buffers on the stack is very coarse, not to mention the fact that it is closed source so extensions are difficult. Valgrind on the other hand is open-source. However it shares the same problem of stack monitoring, not to mention the slowness of execution due to the fact that it runs on a simulated virtual processor.

3.7 Never execute

The Nx-bit [27] represents a mark on certain areas of memory which makes them non-executable, so that the processor will refuse to execute code within those areas. This is relevant to our work because even if this method does not prevent the overwriting of data by a buffer overflow, it will still prevent the execution of any malicious code that is thus injected on the stack.

3.8 Hardware-software hybrid methods

One research work [28] proposes a hardware and software hybrid solution to protect against buffer overflows by introducing new assembly functions. One method is called “Hardware Boundary Check” and it functions by verifying if the target address within a function is equal or larger to the frame pointer; if this is not the case then some reactive measure is taken. The second method introduces two new opcodes, “scall” and “sret”, where scall introduces a signature and sret verifies it.

3.9 Array and pointer boundary checking

A recent investigation [29] uses boundary checking to perform protection against buffer overflows. This solution consists of introducing a new instruction to limit the consumption of system resources by the process.

3.10 Counter-measures using kernel properties

Our work picks up where others left off, mainly the work of another graduate from our university, Benjamin Teissier [30]. He produced a proof-of-concept idea that protects the EIP from modification by any type of attack that may tamper with it such as code injection or buffer overflows. Therefore our work is a direct continuation of his work. He accomplishes the task following these steps:

1) Upon launching a program an interrupt is called to access the kernel.

2) Modules within kernel space are used to patch the corresponding code.

3) Probes within kernel space are then used to manage operations within this environment.

4) Before a call is detected, the EIP is obtained and stored in a variable in kernel space through an injected system call.

5) Once a ret is encountered, the actual EIP on the stack is verified against the value stored in kernel space using yet another injected system call.

While this is in principle sufficient to protect against such types of attacks in 32-bit environments, a few issues arise.

1) It takes a substantial time to perform the patching operation upon the launch of a process. Depending on the size and resources used by the process, this could lead to crashes or timeouts that would leave the process “hanging” indefinitely.

2) The operation takes place in kernel space, and so the size of the text becomes an issue since the space reserved to the kernel is limited.

3) His work does not address any running or active process which in turn leaves the system vulnerable.

4) It only addresses 32-bit environments and yet in 2015 most commercial systems are produced with 64-bit environments.

4 Our Construction

4.1 Challenge

We must define the challenge at a technical level. With our understanding of the functionality of the stack and the theory behind attacks like buffer overflow, we can establish the following:

1) Since the software has vulnerabilities, we cannot fix mistakes associated with the actual coding.

2) We must intercept at execution time the instruction pointers (EIP/RIP respectively) of a process whenever a call is issued.

3) We must validate at execution time that whenever a ret is issued the corresponding return address at the top of the stack has not been modified.

4) We must issue notifications in real-time to the system administrator that a security incident was detected during execution time.

Now that we know what we need to do, we proceed to explain how we do it.

4.2 Procedure

In order to begin our construction we need to establish the overall procedure for our work. Specifically, our solution will:

1) Use ptrace to intercept the opcodes for call and ret.

2) Store the instruction pointer (EIP/RIP respectively) in a buffer upon a call.

3) Detect when a ret is issued.

4) Compare the value at the current stack pointer with the value stored in our buffer.

5) Notify the system administrator should a problem be detected.

4.3 Environment

We now define the environment in which we developed our solution, keeping in mind that the scope of our work will include other GNU/Linux distributions. We define what operating system we will use, as well as the C compiler and python compiler which will be used to code and compile our solution. Therefore we define our platform as follows:

Distribution	Fedora 19
Memory	7.4 GB
Processor	AMD E2-2000 APU with Radeon(tm) HD Graphics x 2
OS type	64-bit
Graphics	Gallium 0.4 on AMD PALM
Gnome	Version 3.8.4
Disk	25.9 GB
C compiler	gcc-4.8.3-7.fc19.x86_64
Python	python-2.7.5-15.fc19.x86_64

Table 2: Development platform

4.4 Solution

Having outlined our overall procedure, as well as the platform we will be using, we proceed to develop our solution using ptrace.

4.4.1 Usage

In order to execute the program in a terminal, we type:

./<launch ctl> <target program>

where **<launch ctl>** is our program (that controls the launching of other programs,) and **<target**

program> is the process to be analyzed. Specific files and examples are detailed in Section 5.3.

4.4.2 Code

First, it is necessary to tell the kernel that a process will be traced. This is accomplished with the PTRACE_TRACEME call whose syntax is:

ptrace(PTRACE_TRACEME, pid,...);

In this call pid is the process id to trace. This is usually followed by an execve() system call which is used to obtain said pid. It is worth mentioning that in our code the pid is a value input by the user and then sent to a function, so we use:

execl(programname, programname, (char *)0);

where execl() is used to obtain the pid which is stored in the variable programname. Once PTRACE_TRACEME is issued, the kernel is aware that the process is being traced and ptrace splits the process into two parts, a parent and a child. This mechanism allows ptrace to hand control back and forth from the parent to the child in order to detect or perform operations as needed. In this case, the child will be executing the process and the parent will check for arguments or look into registers. For our construction the child executes the execve() system call and so hands control over to the parent. Now we must issue a wait() call to stop the execution of the process during the first instruction, we accomplish this in our code with:

wait(&wait_status);

The parameter wait_status is just a variable used to manage the wait signal. It is at this point that we must read the current register values of the process. We accomplish that with PTRACE_GETREGS to obtain the registers and then PTRACE_PEEKTEXT to obtain the values we need. Our implementation of this in our code is as follows: To obtain the registers we do

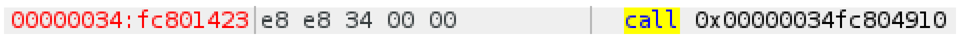
ptrace(PTRACE_GETREGS, child_pid, 0, &regs);

where child_pid is the process being traced. To obtain the instruction pointer for our 64-bit platform we use

ptrace(PTRACE_PEEKTEXT, child_pid, regs.rip, 0);

where child_pid is the process being traced and regs.rip is the instruction pointer for 64-bit architectures. We use temporary variables to store this value.

At this point in our solution we have successfully stopped the execution of the process during run-time, obtained the value for the instruction pointer, and we must now proceed to evaluate if the current instruction being executed is a call or a ret. We accomplish this by taking the hexadecimal value of the instruction and storing it as individual characters in an array. We then evaluate the array by looking for the first two characters and validate if they comprise a call or ret opcode. This is better explained with an example so let us visualize a generic call in order to understand how the instruction is issued:



```
00000034:fc801423 e8 e8 34 00 00        call 0x00000034fc804910
```

Illustration 12: Call execution

In this case, the red hexadecimal characters are the memory address, while the characters next to it (e8 e8 34 00 00) are the hexadecimal characters of the instruction associated to that address, that is, at the instruction pointer. The hand right side of the image explains what those hexadecimal values of the instruction mean. In this case the opcode is e8 and we know that it represents a call. The last set of hexadecimal characters are the address to which the call will jump to. Going back to our solution, what we are essentially doing is taking the hexadecimal values of the instruction, storing them in variables and evaluating if the first two characters are the opcodes for either a call or a ret. We accomplish that with the following code. To look for a call opcode we do:

if (op==0xe8 || op==0x9a || op==0xff || op2 == 0xff00)

The following will determine whether the opcode is a ret code:

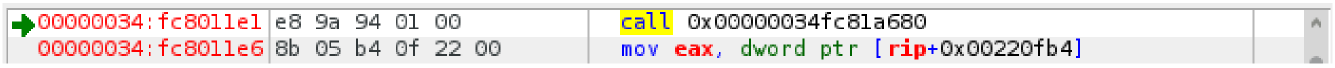
if (op==0xc3 || op==0xcb || op==0xc2 || op==0xca)

At this stage, we must evaluate each condition accordingly. When a call is found, we will obtain the stack pointer and then we store that in a buffer under our control. To obtain the stack pointer for our 64-bit platform we use:

ptrace(PTRACE_PEEKDATA, child_pid, regs.rsp, 0);

where child_pid is the process being traced and regs.rsp is the stack pointer for 64-bit architectures. If on the other hand a ret is detected, then we must check whether the return address on the stack has not been modified. To accomplish this we obtain the current return address from the stack and compare it with the latest value stored in our buffer. If both values are the same, then nothing is wrong and the program continues. However, if the values are different, we then compare the current value in the stack to the value next-to-last in our buffer; if they still do not match, we then compare it to the next value in our buffer, and so on. It is important to note that our buffer is effectively a stack, which stores the latest value on the top which is then compared with the value of the normal stack when a ret happens. However since we are dealing with thousands of instructions being executed, a buffer is necessary in order to store previous values from previous call opcodes. In addition, we also intercept jmp opcodes in order to simplify the issue of the rew.x + ff prefix, which effectively leaves us intercepting any opcode preceded by any two values followed by an ff. This will all be detailed in Section 6.1.1, but for now we note that a consequence of this processing is that it is no longer enough to verify the top of the buffer, but we need to go deeper in the buffer with our comparison instead. In short, our buffer is a second stack that we use to store the values we are interested in, which are the call and ret stack pointers. This is better explained with an example.

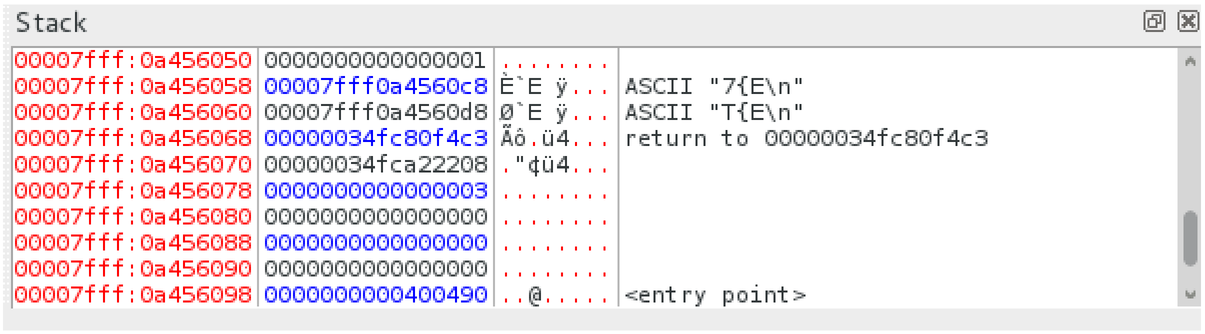
Suppose that the execution of a process eventually reaches a call opcode:



```
00000034:fc8011e1 e8 9a 94 01 00     call 0x00000034fc81a680
00000034:fc8011e6 8b 05 b4 0f 22 00  mov eax, dword ptr [rip+0x00220fb4]
```

Illustration 13: Call execution

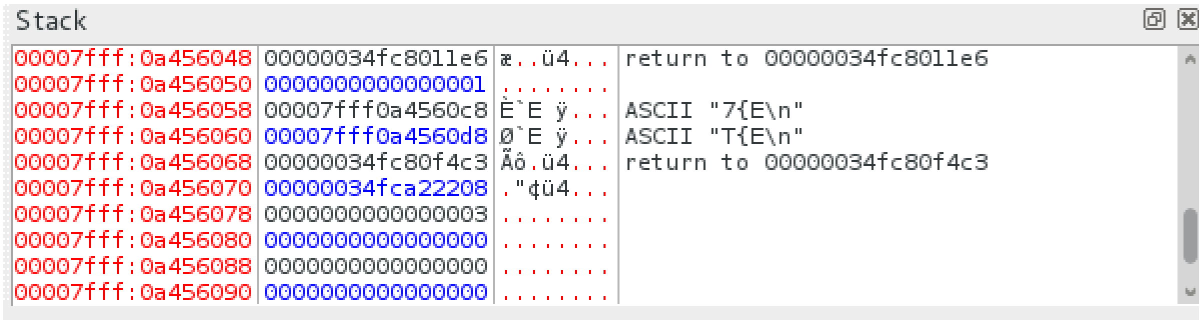
In our current execution flow, we have stopped at a call associated to the address 34fc8011e1 and we can see that the call will jump to the address 34fc81a680. However, we know that at some point in the execution a ret will be issued to return to the address 34fc8011e6, which is the next address following the call. Now we will visualize the stack:



```
Stack
00007fff:0a456050 0000000000000001 ........
00007fff:0a456058 00007fff0a4560c8 È`E ÿ... ASCII "7{E\n"
00007fff:0a456060 00007fff0a4560d8 Ø`E ÿ... ASCII "T{E\n"
00007fff:0a456068 00000034fc80f4c3 Ãô.ü4... return to 00000034fc80f4c3
00007fff:0a456070 00000034fca22208 ."¢ü4...
00007fff:0a456078 0000000000000003 ........
00007fff:0a456080 0000000000000000 ........
00007fff:0a456088 0000000000000000 ........
00007fff:0a456090 0000000000000000 ........
00007fff:0a456098 0000000000400490 ..@..... <entry point>
```

Illustration 14: Stack viewer

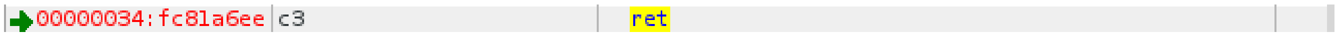
As we can see, the stack is empty since the program has just begun its execution. We will now continue the execution and since we already know that the stack pointer pushed by the call will point to the next address right after call (34fc8011e6) we will expect that to appear on our stack:



```
00007fff:0a456048 00000034fc8011e6 æ..ü4... return to 00000034fc8011e6
00007fff:0a456050 0000000000000001 ........
00007fff:0a456058 00007fff0a4560c8 È`E ÿ... ASCII "7{E\n"
00007fff:0a456060 00007fff0a4560d8 Ø`E ÿ... ASCII "T{E\n"
00007fff:0a456068 00000034fc80f4c3 Ãô.ü4... return to 00000034fc80f4c3
00007fff:0a456070 00000034fca22208 ."¢ü4...
00007fff:0a456078 0000000000000003 ........
00007fff:0a456080 0000000000000000 ........
00007fff:0a456088 0000000000000000 ........
00007fff:0a456090 0000000000000000 ........
```

Illustration 15: Stack viewer

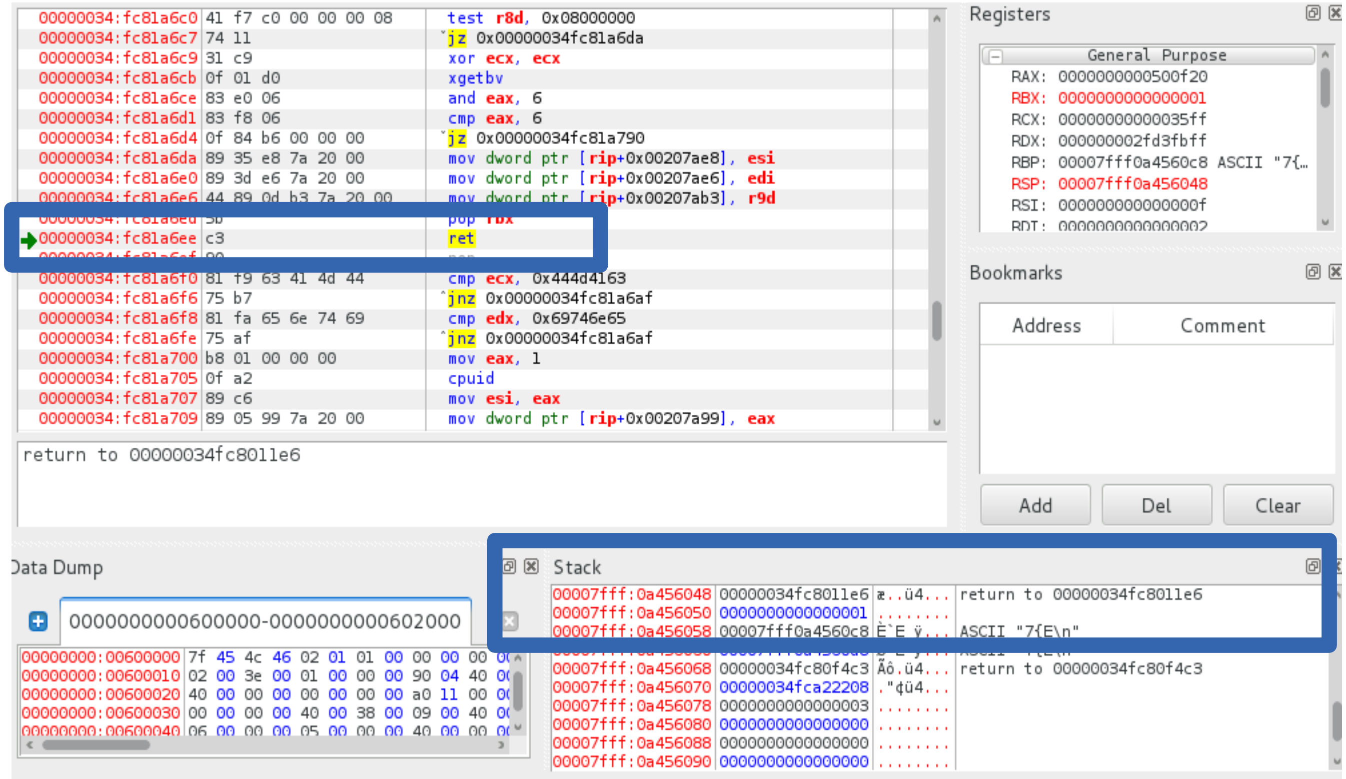
Indeed, the stack now includes the address 34fc8011e6, as expected. Going back to our solution, it is at this point that we store the return address on our buffer and let the process continue. Now we will carry on with the execution flow until we find the first ret:



```
00000034:fc81a6ee c3                ret
```

Illustration 16: Ret execution

We now check to see if the stack value points to the expected address (34fc8011e6):



```
00000034:fc81a6c0 41 f7 c0 00 00 00 08  test r8d, 0x08000000
00000034:fc81a6c7 74 11                 jz 0x00000034fc81a6da
00000034:fc81a6c9 31 c9                 xor ecx, ecx
00000034:fc81a6cb 0f 01 d0              xgetbv
00000034:fc81a6ce 83 e0 06              and eax, 6
00000034:fc81a6d1 83 f8 06              cmp eax, 6
00000034:fc81a6d4 0f 84 b6 00 00 00     jz 0x00000034fc81a790
00000034:fc81a6da 89 35 e8 7a 20 00     mov dword ptr [rip+0x00207ae8], esi
00000034:fc81a6e0 89 3d e6 7a 20 00     mov dword ptr [rip+0x00207ae6], edi
00000034:fc81a6e6 44 89 0d b3 7a 20 00  mov dword ptr [rip+0x00207ab3], r9d
00000034:fc81a6ed 5b                    pop rbx
00000034:fc81a6ee c3                    ret
00000034:fc81a6ef 90                    nop
00000034:fc81a6f0 81 f9 63 41 4d 44     cmp ecx, 0x444d4163
00000034:fc81a6f6 75 b7                 jnz 0x00000034fc81a6af
00000034:fc81a6f8 81 fa 65 6e 74 69     cmp edx, 0x69746e65
00000034:fc81a6fe 75 af                 jnz 0x00000034fc81a6af
00000034:fc81a700 b8 01 00 00 00        mov eax, 1
00000034:fc81a705 0f a2                 cpuid
00000034:fc81a707 89 c6                 mov esi, eax
00000034:fc81a709 89 05 99 7a 20 00     mov dword ptr [rip+0x00207a99], eax

return to 00000034fc8011e6
```

```
RAX: 0000000000500f20
RBX: 0000000000000001
RCX: 00000000000035ff
RDX: 000000002fd3fbff
RBP: 00007fff0a4560c8 ASCII "7{…
RSP: 00007fff0a456048
RSI: 000000000000000f
RDI: 0000000000000002
```

```
0000000000600000-0000000000602000

00000000:00600000 7f 45 4c 46 02 01 01 00 00 00 00 00
00000000:00600010 02 00 3e 00 01 00 00 00 90 04 40 00
00000000:00600020 40 00 00 00 00 00 00 00 a0 11 00 00
00000000:00600030 00 00 00 00 40 00 38 00 09 00 40 00
00000000:00600040 06 00 00 00 05 00 00 00 40 00 00 00
```

```
00007fff:0a456048 00000034fc8011e6 æ..ü4... return to 00000034fc8011e6
00007fff:0a456050 0000000000000001 ........
00007fff:0a456058 00007fff0a4560c8 È`E ÿ... ASCII "7{E\n"
00007fff:0a456060 00007fff0a4560d8 Ø`E ÿ... ASCII "T{E\n"
00007fff:0a456068 00000034fc80f4c3 Ãô.ü4... return to 00000034fc80f4c3
00007fff:0a456070 00000034fca22208 ."¢ü4...
00007fff:0a456078 0000000000000003 ........
00007fff:0a456080 0000000000000000 ........
00007fff:0a456088 0000000000000000 ........
00007fff:0a456090 0000000000000000 ........
```

Illustration 17: Ret and stack validation

The ret value is the correct one and thus the program continues on as usual. Going back to our solution, it is that this point where we check whether the current value on the stack matches the value in our buffer.

Once the comparison between the current value on the stack and the values in our buffer is complete, if the current value in the stack is not found then something modified it on the stack and therefore we assume there has been an attack such as a buffer overflow. If this is the case, we now issue a real-time message to the system administrator and kill the process. To accomplish this we developed a server script and a client script in python to handle the real-time messaging.

The server script will be executing at the system administrator's computer actively listening for messages at a port of his or her choice. In our code we generate a network socket and bind it to port 5555. The code to accomplish this is:

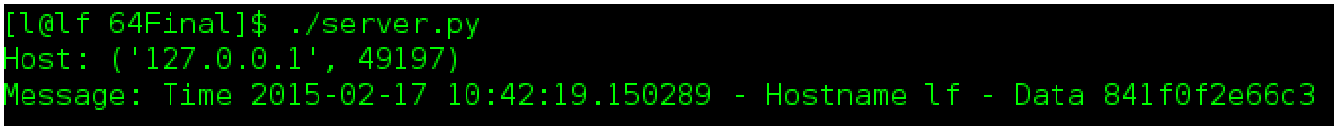
```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
s.listen(1)
conn, addr = s.accept()
```

The client script operates in a similar manner by generating a socket, binding it to a port and sending a message through the network in plain text. The message includes the network address of the host, the current system time of the host, the hostname, and the instruction that was executed when the incident occurred (in hexadecimal). While an argument could be made regarding the actual necessity of the message, it is important to remember what we established at the beginning of this thesis: we seek to develop a solution that not only automatically detects problems, but also issues notifications that contains precise information. In this case the fact that we are sending valuable information like time, hostname, network address and even more so, the actual instruction that caused the problem, can potentially save the user time and other resources should the need to perform forensic analysis arise. The code to accomplish all of this is:

```
 msg = 'Time %s - Hostname %s - Data %s\n'
(datetime.datetime.now(),socket.gethostname(),dat)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
s.send(msg)
```

s.close()

This is an sample message received by the system administrator, should a problem be detected on a remote machine:



```
[l@lf 64Final]$ ./server.py
Host: ('127.0.0.1', 49197)
Message: Time 2015-02-17 10:42:19.150289 - Hostname lf - Data 841f0f2e66c3
```

Illustration 18: Server message

At this point in our solution, we have successfully stopped the execution of the process during run time, obtained the value for the instruction pointer, and evaluated whether the current instruction being executed is a call or a ret. If a call was found then the corresponding value is stored in a buffer, while if a ret was found then we compare the current value in the stack to the values stored in the buffer; should a discrepancy be detected, our solution automatically issues a message with pertinent information to the system administrator over the network. It is now necessary to perform this during every instruction executed in order to fully protect the process. To accomplish this, we use PTRACE_SINGLESTEP to advance the process to the next instruction. The code for this is:

ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0)

Finally, the child hands control back to the parent and the parent waits for notification from the kernel with a wait() call, thus completing the cycle. In conclusion, this whole sequence will take place during every instruction in the process which translates into a successful, working solution that fulfills the objectives we had set out to accomplish.

4.5 Faster

As stated in the previous section, we have successfully developed a working solution that accomplishes the objectives we had set out to achieve. However there is still one issue, namely the time it takes to analyze the process. It all boils down to the buffer and the use of ptrace; while we cannot optimize ptrace since it is built into the system, it is still the case that the bigger the process we are analyzing, the bigger the buffer will need to be and therefore the time needed to perform our operations will increase accordingly. For these reasons we propose a solution to this issue taking into consideration a couple of things:

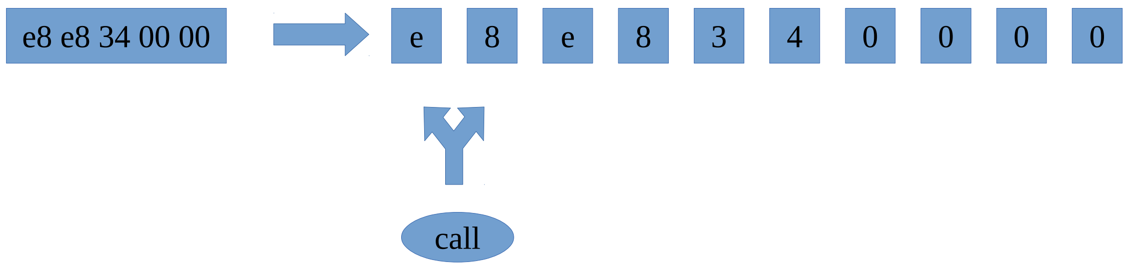
1) Checking every single call/ret greatly increases the time requirements, so we would like to come up

with an alternative.

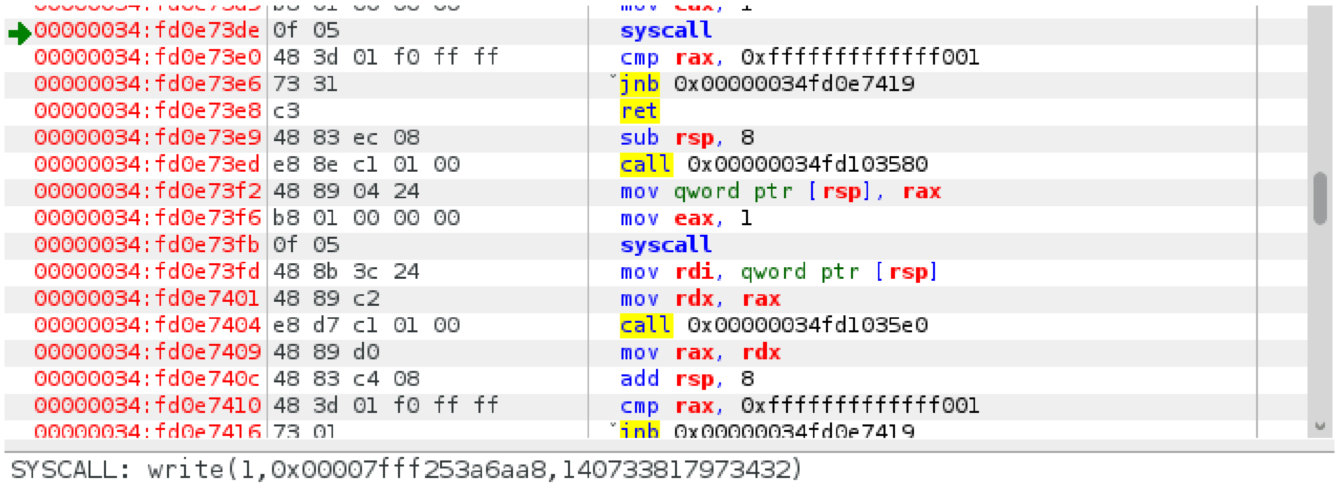
2) If we assume that an attack is taking place, we can argue that a write() system call could potentially be issued.

3) Therefore if we perform the same operations as before but only whenever a write() system call is detected, we can successfully detect attacks while greatly decreasing the time overhead.

Thus we also developed a solution based on these considerations. Our reasoning behind the concept of using write system calls to detect stack buffer overflows is that in practice, while monitoring the execution of overflowing processes, we detected that system calls were issued at critical points in the buffer overflow exploit. More specifically we identified that such calls were in place when the last ret was found (the place of the detection of the overflow). In order to detect opcodes, we generated files that would serve as storage for the instructions in hexadecimal, which are then read back into an array and split into individual characters. The first two characters are then used to evaluate whether a call or ret opcode was detected:



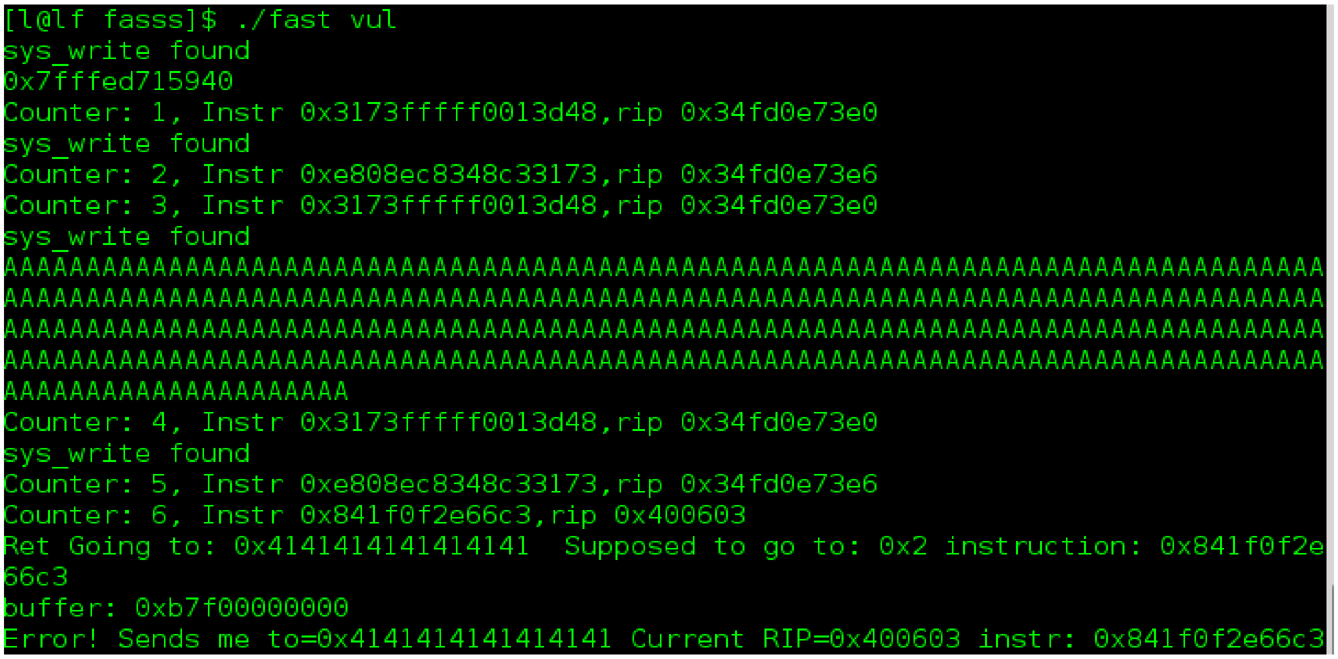
In order to exemplify this, we will validate our reasoning based on the actual execution of our exploit:



```
00000034:fd0e73de 0f 05              syscall
00000034:fd0e73e0 48 3d 01 f0 ff ff  cmp rax, 0xfffffffffffff001
00000034:fd0e73e6 73 31              jnb 0x00000034fd0e7419
00000034:fd0e73e8 c3                 ret
00000034:fd0e73e9 48 83 ec 08        sub rsp, 8
00000034:fd0e73ed e8 8e c1 01 00     call 0x00000034fd103580
00000034:fd0e73f2 48 89 04 24        mov qword ptr [rsp], rax
00000034:fd0e73f6 b8 01 00 00 00     mov eax, 1
00000034:fd0e73fb 0f 05              syscall
00000034:fd0e73fd 48 8b 3c 24        mov rdi, qword ptr [rsp]
00000034:fd0e7401 48 89 c2           mov rdx, rax
00000034:fd0e7404 e8 d7 c1 01 00     call 0x00000034fd1035e0
00000034:fd0e7409 48 89 d0           mov rax, rdx
00000034:fd0e740c 48 83 c4 08        add rsp, 8
00000034:fd0e7410 48 3d 01 f0 ff ff  cmp rax, 0xfffffffffffff001
00000034:fd0e7416 73 01              jnb 0x00000034fd0e7419
SYSCALL: write(1,0x00007fff253a6aa8,140733817973432)
```

Illustration 19: System call write

At this stage in the execution we come across a write system call, which is identified in the area below the image. In order to better understand how system calls are issued and visualize how this is relevant to our work, we consider how many system calls were detected during the execution of our code:



```
[l@lf fasss]$ ./fast vul
sys_write found
0x7fffed715940
Counter: 1, Instr 0x3173fffff0013d48,rip 0x34fd0e73e0
sys_write found
Counter: 2, Instr 0xe808ec8348c33173,rip 0x34fd0e73e6
Counter: 3, Instr 0x3173fffff0013d48,rip 0x34fd0e73e0
sys_write found
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
Counter: 4, Instr 0x3173fffff0013d48,rip 0x34fd0e73e0
sys_write found
Counter: 5, Instr 0xe808ec8348c33173,rip 0x34fd0e73e6
Counter: 6, Instr 0x841f0f2e66c3,rip 0x400603
Ret Going to: 0x4141414141414141  Supposed to go to: 0x2 instruction: 0x841f0f2e
66c3
buffer: 0xb7f00000000
Error! Sends me to=0x4141414141414141 Current RIP=0x400603 instr: 0x841f0f2e66c3
```

Illustration 20: System calls

We note that the counter represents how many times our code analyzed an instruction, and thus we can identity a pattern. Specifically, the same system calls are used again and again during the execution of the process and each time the values may or may not change. This is visualized by the fact that we can see the same two RIPs being called at different times (0x34fd0e73e0 and 0x34fd0e73e6) which we can validate against the previous image of the write system call as being the next immediate address after that write system call. In other words, the same system call happened twice during execution but with different instructions each time. Also note the error message displayed when the overflow is detected (0x41414141414...).

Overall, the detection of the last ret after a write system call allows us to verify that since there is no associated call stack pointer in our buffer, an anomaly is detected. No other calls participate in this decision (meaning that all the other matching rets are correct). A message is then issued with the information of the address that the stack buffer overflow pushed incorrect information on the stack which in turn altered the control flow of the program ("Sends me to=0x41414141414..."). We also have the detected instruction, which we can verify that it ends in the value of c3 which is the opcode for ret. Therefore we conclude that for our example detection of write system calls is successful in the detection of stack buffer overflows. We will detail this further in Section 6.1.

In practical terms, the mechanisms between the normal and fast versions are effectively the same; however the change comes via this piece of code which detects only write() system calls:

if((regs.orig_rax == SYS_write))

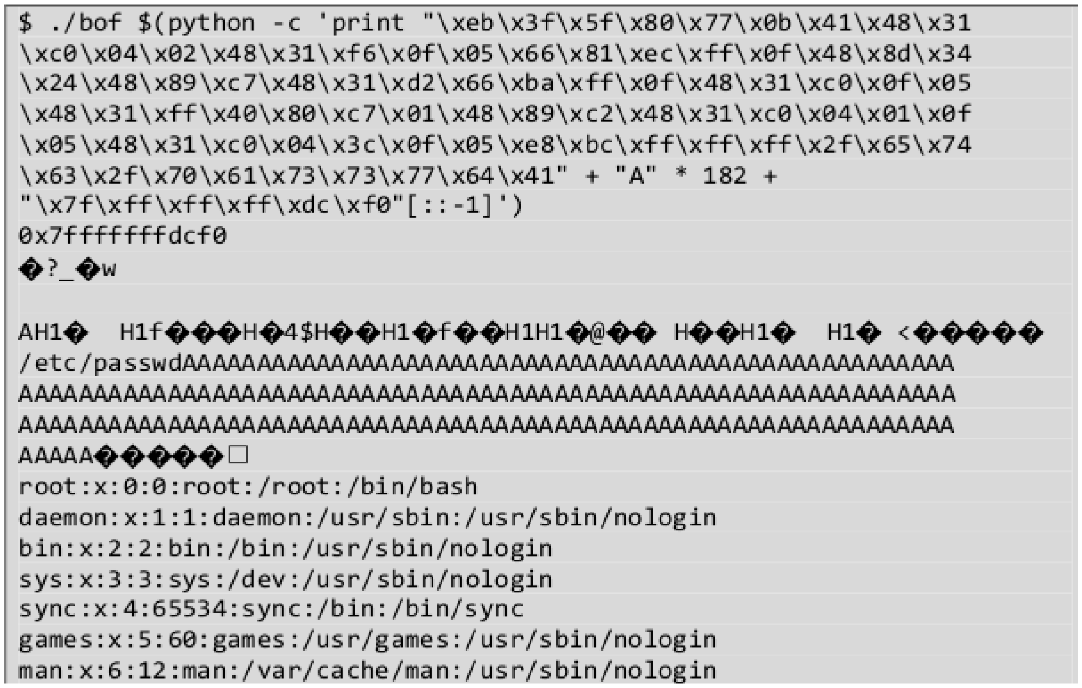
4.6 32-bit

The 32-bit version operates in exactly the same manner. However the code had to be adapted to the proper syntax. For example, regs.rip for 64-bit would be regs.eip for 32-bit. An important thing to note is that while observing our “if” conditions in the code we established that the following two characters in the array for the ret opcode of c2 should be any two characters. This is because in practice we noticed that values with no associated call actually get pushed onto the stack. It is also important to note that the fast version does not work on 32-bit architectures; we will discuss limitations in further sections.

5 Evaluation

5.1 Exploits

In order to show that our solution works we will use a published code by the Mr.Un1k0d3r RingZer0 Team that exploits a buffer overflow to execute a piece of shell code which in turn obtains the password files in a Linux system. The following is an example of the exploit after successfully overflowing the code:



```
$ ./bof $(python -c 'print "\xeb\x3f\x5f\x80\x77\x0b\x41\x48\x31
\xc0\x04\x02\x48\x31\xf6\x0f\x05\x66\x81\xec\xff\x0f\x48\x8d\x34
\x24\x48\x89\xc7\x48\x31\xd2\x66\xba\xff\x0f\x48\x31\xc0\x0f\x05
\x48\x31\xff\x40\x80\xc7\x01\x48\x89\xc2\x48\x31\xc0\x04\x01\x0f
\x05\x48\x31\xc0\x04\x3c\x0f\x05\xe8\xbc\xff\xff\xff\x2f\x65\x74
\x63\x2f\x70\x61\x73\x73\x77\x64\x41" + "A" * 182 +
"\x7f\xff\xff\xff\xdc\xf0"[::-1]')
0x7fffffffdcf0
�?_�w

AH1�  H1f���H�4$H��H1�f��H1H1�@�� H��H1�  H1� <������
/etc/passwdAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA������□
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
```

Illustration 21: Buffer overflow exploit

The justification for the use of this particular exploit is that this is a textbook example of a buffer overflow, which is well documented in both our Preliminaries section as well as in the actual source [31]. We can also visualize the code and the hexadecimal values step by step during normal run-time as well as during the overflow and payload injection. This code was also published in the exploit database website [31] which is arguably one of the best sources of old and current exploits in the wild. Thus we argue that our example comes from a well-known source, it is documented in detail down to hexadecimal values and source code, and has a textbook behavior. We can therefore safely assume that other stack buffer overflows will follow the same fundamental mechanism.

We will also manually inject code into the process in order to simulate code injection in real time in any given process at any given time during execution. We accomplish this by using ptrace itself:

```
ptrace(PTRACE_POKEDATA, child_pid,regs.rsp, 333);
```

where we inject the value 333 into the current position in the stack. With these 2 scenarios we will demonstrate how our solution successfully performs the required tasks.

5.2 Benchmarking

As previously stated time is an issue, so we will measure how long it takes to perform the operations. We accomplish this by including the standard clock functions for C in our code:

```
#include <time.h>

clock_t t;

t=clock();

...

printf("EXTERNAL: Clicks: %d (%f seconds)\n",t,((float)t)/CLOCKS_PER_SEC);
```

The time displayed is based on the internal clock of the processor, not the actual time as we conventionally measure it.

5.3 Testing

We will now present our solution working on three different distributions of GNU/Linux. It is important to note that these distributions were installed on virtual machines and therefore the hardware specifications are the same for all of them. We will also define the names of the executables as to allow for a clearer understanding of the screenshots:

Name	Description
64full	Compiled from 64full.c, it is the code that single-steps through the process at every instruction executed for 64-bit machines. (113 lines of code)
64fullinj	Compiled from 64fullinj.c, it is the code that singlesteps through the process at every instruction executed for 64-bit machines with code injection. (113 lines of code)
64fast	Compiled from 64fast.c, it is the code that single-steps through the process whenever a write() system call is detected for 64-bit machines. (107 lines of code)
64fastinj	Compiled from 64fastinj.c, it is the code that single-steps through the process whenever a write() system call is detected for 64-bit machines with code injection. (107 lines of code)
32full	Compiled from 32full.c, it is the code that single-steps through the process at every instruction executed for 32-bit machines. (113 lines of code)
32inj	Compiled from 32inj.c, it is the code that single-steps through the process at every instruction executed for 32-bit machines with code injection. (113 lines of code)
Loop	Compiled from loop.c, it is the code for a simple loop used for testing with 64-bit machines.
Loo	Compiled from loo.c, it is the code for a simple loop used for testing with 32-bit machines.
Vul	Compiled from vul.c, it is the code that exploits a buffer overflow obtained from the internet, used for testing with 64-bit machines.
Vuln	Compiled from vuln.c, it is the code that exploits a buffer overflow obtained from the internet, used for testing with 32-bit machines.
Client.py	Python script for the client messaging socket. (14 lines of code)
Server.py	Python script for the server messaging socket. (19 lines of code)

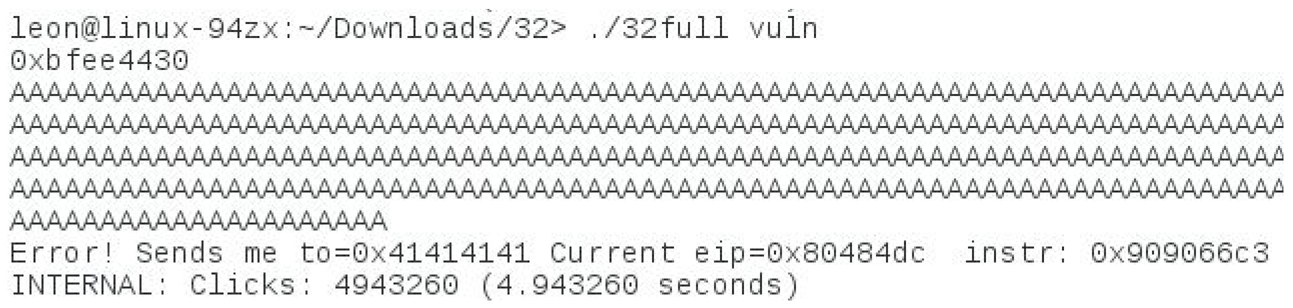
Table 3: Files

5.3.1 OpenSUSE

Distribution	OpenSUSE 13.2 (Harlequin) (i568) 32-bit
Memory	2.0 GB
Processor	Intel Core i3-3220 CPU @ 3.30 GHz
OS type	32-bit
Graphics	Gallium 0.4 on llvmpipe (LLVM 3.5, 128 bits)
Gnome	Version 3.14.1
Virtualization	Oracle (Virtualbox 4.3.20 r96997)
C compiler	Gcc 4.8.3 20140627
Python	2.7.8

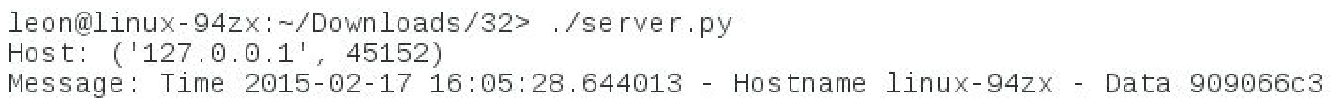
Table 4: OpenSUSE 32-bit

For the 32-bit version of OpenSUSE, executing the code of the buffer overflow [31] we present the screenshots of the successful detection of a buffer overflow with the corresponding message received by the system administrator:



```
leon@linux-94zx:~/Downloads/32> ./32full vuln
0xbfee4430
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
Error! Sends me to=0x41414141 Current eip=0x80484dc  instr: 0x909066c3
INTERNAL: Clicks: 4943260 (4.943260 seconds)
```

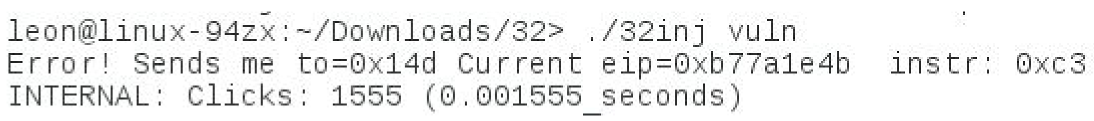
Illustration 22: Buffer overflow



```
leon@linux-94zx:~/Downloads/32> ./server.py
Host: ('127.0.0.1', 45152)
Message: Time 2015-02-17 16:05:28.644013 - Hostname linux-94zx - Data 909066c3
```

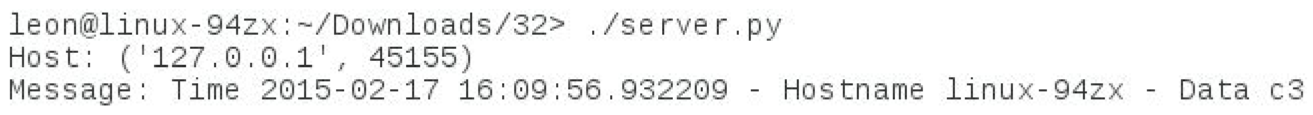
Illustration 23: Server message

For the 32-bit version of OpenSUSE, executing the code for the injection (Section 8.3), we present the screenshots of the successful detection of a code injection with the corresponding message received by the system administrator:



```
leon@linux-94zx:~/Downloads/32> ./32inj vuln
Error! Sends me to=0x14d Current eip=0xb77a1e4b  instr: 0xc3
INTERNAL: Clicks: 1555 (0.001555_seconds)
```

Illustration 24: Injection



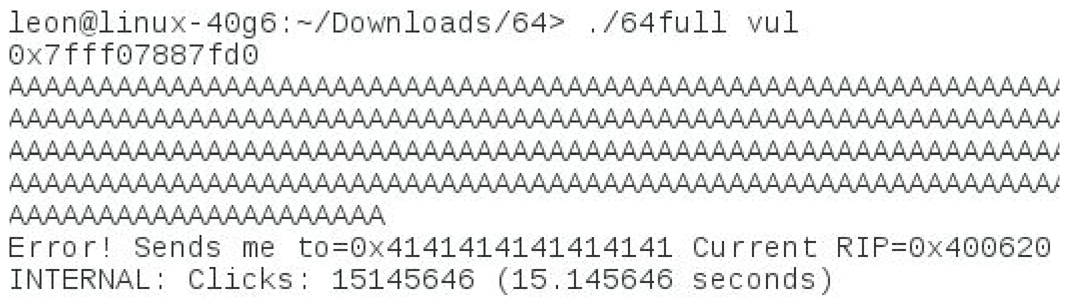
```
leon@linux-94zx:~/Downloads/32> ./server.py
Host: ('127.0.0.1', 45155)
Message: Time 2015-02-17 16:09:56.932209 - Hostname linux-94zx - Data c3
```

Illustration 25: Server message

Distribution	OpenSUSE 13.2 (Harlequin) (x86_64) 64-bit
Memory	2.0 GB
Processor	Intel Core i3-3220 CPU @ 3.30 GHz
OS type	64-bit
Graphics	Gallium 0.4 on llvmpipe (LLVM 3.5, 128 bits)
Gnome	Version 3.14.1
Virtualization	Oracle (Virtualbox 4.3.20 r96997)
C compiler	Gcc 4.8.3 20140627
Python	2.7.8

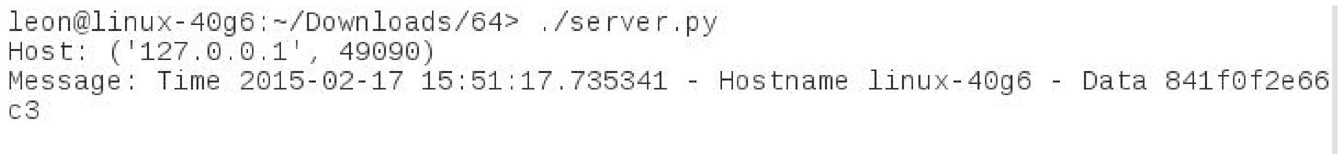
Table 5: OpenSUSE 64-bit

For the 64-bit version of OpenSUSE, executing the code of the buffer overflow [31] we present the screenshots of the successful detection of a buffer overflow with the corresponding message received by the system administrator:



```
leon@linux-40g6:~/Downloads/64> ./64full vul
0x7fff07887fd0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
Error! Sends me to=0x4141414141414141 Current RIP=0x400620
INTERNAL: Clicks: 15145646 (15.145646 seconds)
```

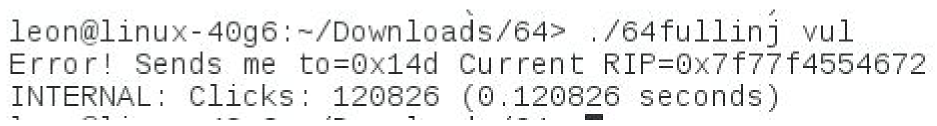
Illustration 26: Buffer overflow



```
leon@linux-40g6:~/Downloads/64> ./server.py
Host: ('127.0.0.1', 49090)
Message: Time 2015-02-17 15:51:17.735341 - Hostname linux-40g6 - Data 841f0f2e66
c3
```

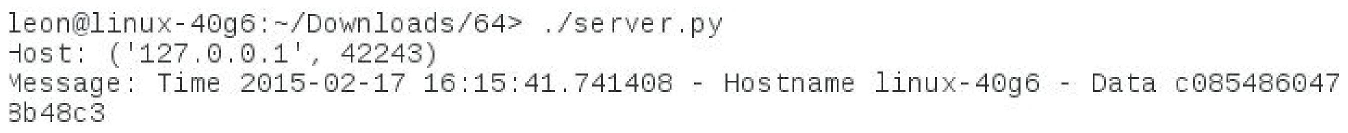
Illustration 27: Server message

For the 64-bit version of OpenSUSE, executing the code for the injection (Section 8.2), we present the screenshots of the successful detection of a code injection with the corresponding message received by the system administrator:



```
leon@linux-40g6:~/Downloads/64> ./64fullinj vul
Error! Sends me to=0x14d Current RIP=0x7f77f4554672
INTERNAL: Clicks: 120826 (0.120826 seconds)
```

Illustration 28: Injection



```
leon@linux-40g6:~/Downloads/64> ./server.py
Host: ('127.0.0.1', 42243)
Message: Time 2015-02-17 16:15:41.741408 - Hostname linux-40g6 - Data c085486047
3b48c3
```

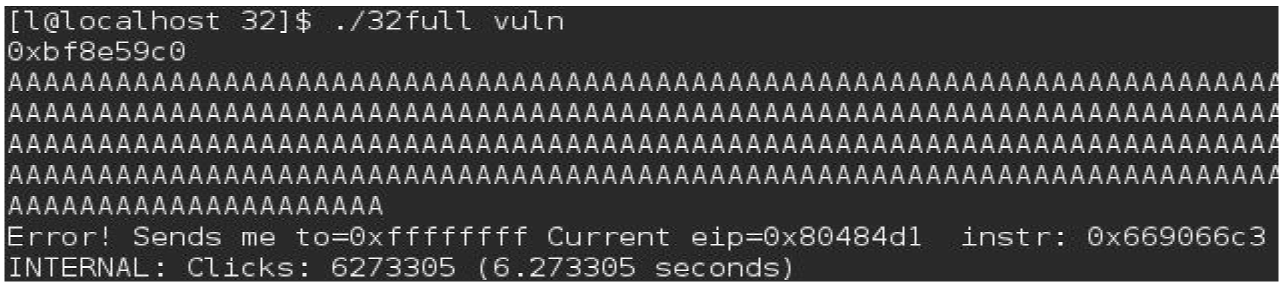
Illustration 29: Server message

5.3.2 Fedora

Distribution	Fedora 21
Memory	2.0 GB
Processor	Intel Core i3-3220 CPU @ 3.30 GHz
OS type	32-bit
Graphics	Gallium 0.4 on llvmpipe (LLVM 3.5, 128 bits)
Gnome	Version 3.14.2
Virtualization	Oracle (Virtualbox 4.3.20 r96997)
C compiler	Gcc 4.9.2 20141101
Python	2.7.8

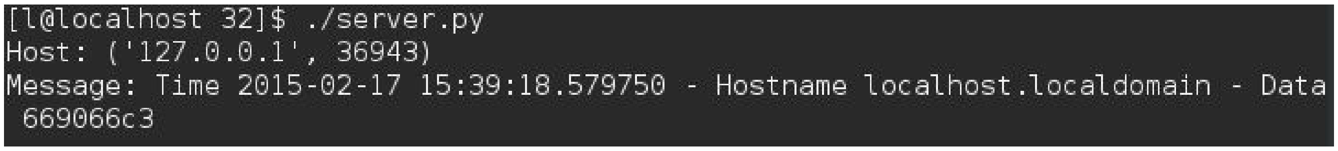
Table 6: Fedora 32-bit

For the 32-bit version of Fedora, executing the code of the buffer overflow [31] we present the screenshots of the successful detection of a buffer overflow with the corresponding message received by the system administrator:



```
[l@localhost 32]$ ./32full vuln
0xbf8e59c0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA
Error! Sends me to=0xffffffff Current eip=0x80484d1  instr: 0x669066c3
INTERNAL: Clicks: 6273305 (6.273305 seconds)
```

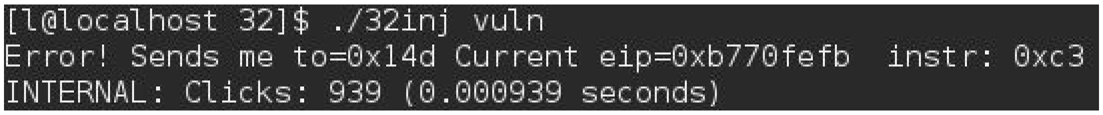
Illustration 30: Buffer overflow



```
[l@localhost 32]$ ./server.py
Host: ('127.0.0.1', 36943)
Message: Time 2015-02-17 15:39:18.579750 - Hostname localhost.localdomain - Data
 669066c3
```

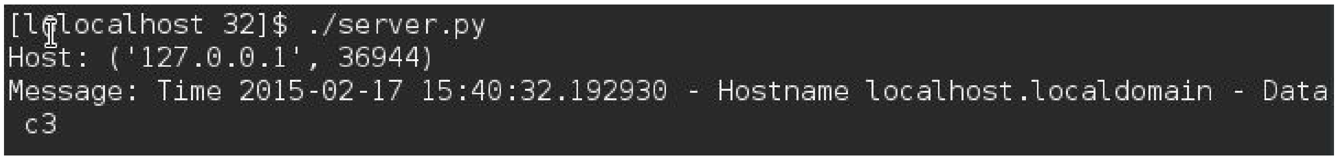
Illustration 31: Server message

For the 32-bit version of Fedora, executing the code for the injection (Section 8.3), we present the screenshots of the successful detection of a code injection with the corresponding message received by the system administrator:



```
[l@localhost 32]$ ./32inj vuln
Error! Sends me to=0x14d Current eip=0xb770fefb  instr: 0xc3
INTERNAL: Clicks: 939 (0.000939 seconds)
```

Illustration 32: Injection



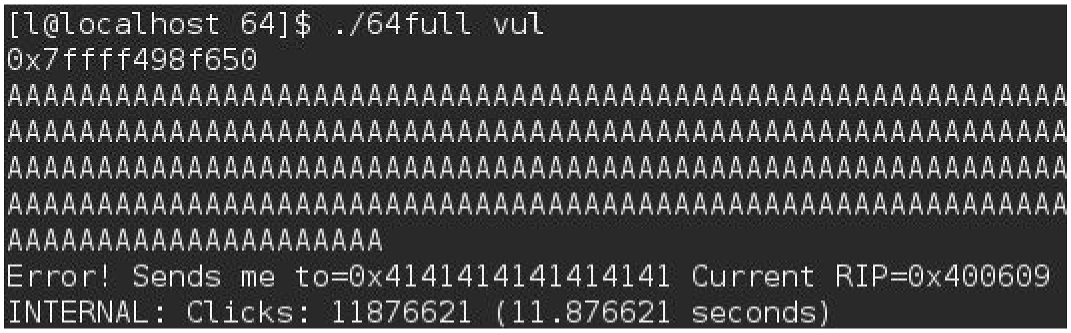
```
[l@localhost 32]$ ./server.py
Host: ('127.0.0.1', 36944)
Message: Time 2015-02-17 15:40:32.192930 - Hostname localhost.localdomain - Data
 c3
```

Illustration 33: Server message

Distribution	Fedora 21
Memory	2.0 GB
Processor	Intel Core i3-3220 CPU @ 3.30 GHz
OS type	64-bit
Graphics	Gallium 0.4 on llvmpipe (LLVM 3.5, 128 bits)
Gnome	Version 3.14.1
Virtualization	Oracle (Virtualbox 4.3.20 r96997)
C compiler	Gcc 4.9.2 20141101
Python	2.7.8

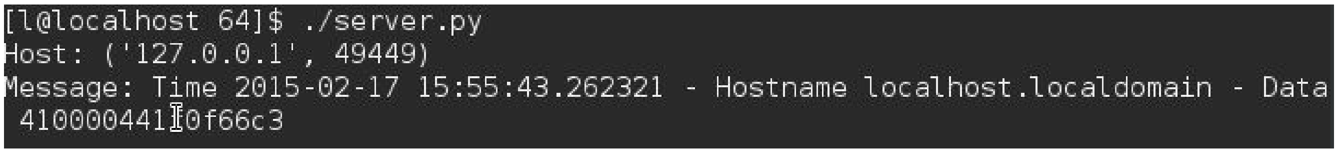
Table 7: Fedora 64-bit

For the 64-bit version of Fedora, executing the code of the buffer overflow [31] we present the screenshots of the successful detection of a buffer overflow with the corresponding message received by the system administrator:



```
[l@localhost 64]$ ./64full vul
0x7ffff498f650
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
Error! Sends me to=0x4141414141414141 Current RIP=0x400609
INTERNAL: Clicks: 11876621 (11.876621 seconds)
```

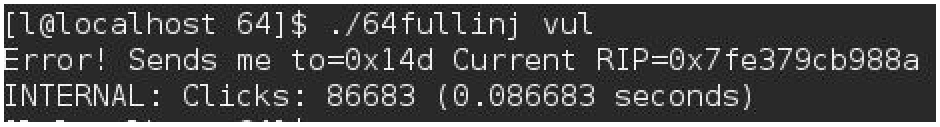
Illustration 34: Buffer overflow



```
[l@localhost 64]$ ./server.py
Host: ('127.0.0.1', 49449)
Message: Time 2015-02-17 15:55:43.262321 - Hostname localhost.localdomain - Data
 410000441f0f66c3
```

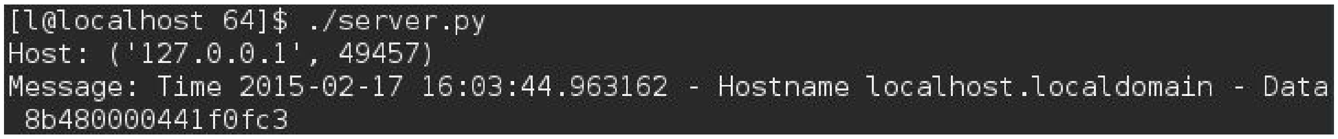
Illustration 35: Server message

For the 64-bit version of Fedora, executing the code for the injection (Section 8.2), we present the screenshots of the successful detection of a code injection with the corresponding message received by the system administrator:



```
[l@localhost 64]$ ./64fullinj vul
Error! Sends me to=0x14d Current RIP=0x7fe379cb988a
INTERNAL: Clicks: 86683 (0.086683 seconds)
```

Illustration 36: Injection



```
[l@localhost 64]$ ./server.py
Host: ('127.0.0.1', 49457)
Message: Time 2015-02-17 16:03:44.963162 - Hostname localhost.localdomain - Data
 8b480000441f0fc3
```

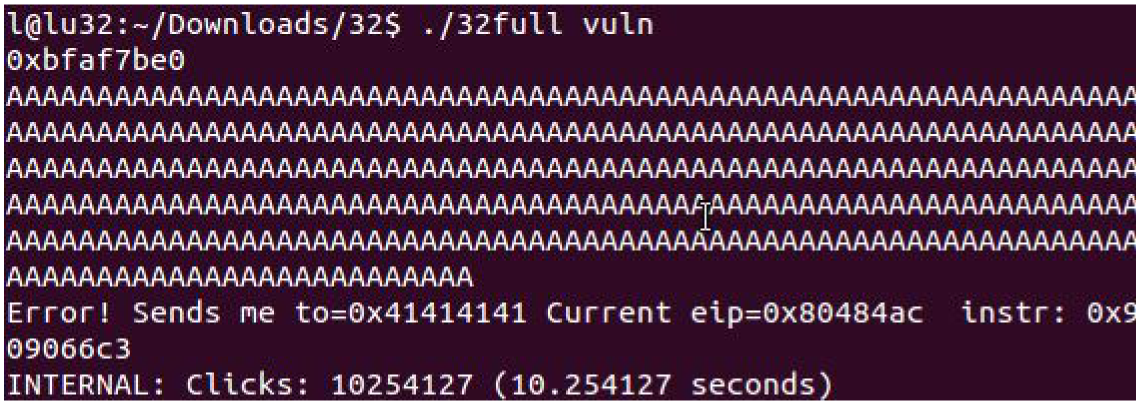
Illustration 37: Server message

5.3.3 Ubuntu

Distribution	Ubuntu 14.04 LTS
Memory	2.0 GB
Processor	Intel Core i3-3220 CPU @ 3.30 GHz
OS type	32-bit
Graphics	Gallium 0.4 on llvmpipe (LLVM 3.5, 128 bits)
Gnome	Gnome-shell not installed
Virtualization	Oracle (Virtualbox 4.3.20 r96997)
C compiler	Gcc 4.8.2
Python	2.7.6

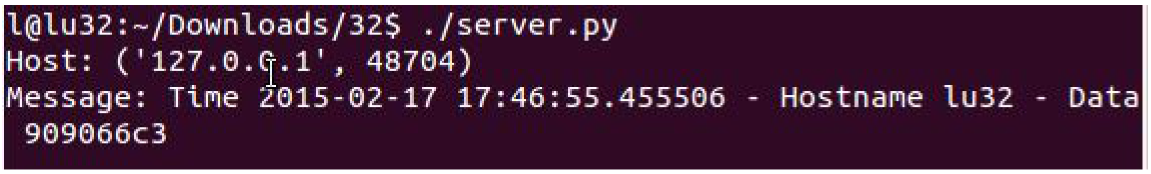
Table 8: Ubuntu 32-bit

For the 32-bit version of Ubuntu, executing the code of the buffer overflow [31] we present the screenshots of the successful detection of a buffer overflow with the corresponding message received by the system administrator:



```
l@lu32:~/Downloads/32$ ./32full vuln
0xbfaf7be0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA
Error! Sends me to=0x41414141 Current eip=0x80484ac  instr: 0x9
09066c3
INTERNAL: Clicks: 10254127 (10.254127 seconds)
```

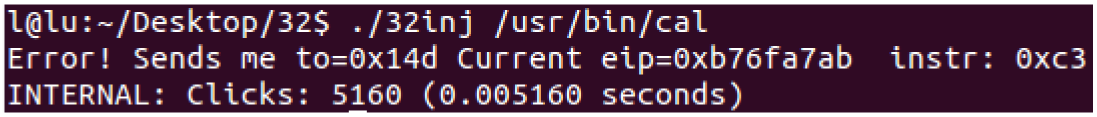
Illustration 38: Buffer overflow



```
l@lu32:~/Downloads/32$ ./server.py
Host: ('127.0.0.1', 48704)
Message: Time 2015-02-17 17:46:55.455506 - Hostname lu32 - Data
 909066c3
```

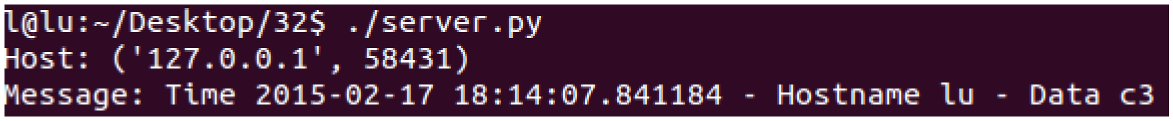
Illustration 39: Server message

For the 32-bit version of Ubuntu, executing the code for the injection (Section 8.3), we present the screenshots of the successful detection of a code injection with the corresponding message received by the system administrator:



```
l@lu:~/Desktop/32$ ./32inj /usr/bin/cal
Error! Sends me to=0x14d Current eip=0xb76fa7ab  instr: 0xc3
INTERNAL: Clicks: 5160 (0.005160 seconds)
```

Illustration 40: Injection



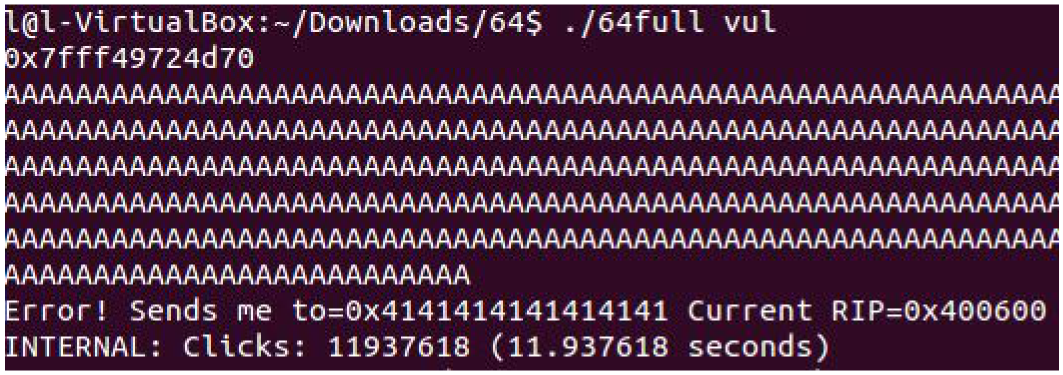
```
l@lu:~/Desktop/32$ ./server.py
Host: ('127.0.0.1', 58431)
Message: Time 2015-02-17 18:14:07.841184 - Hostname lu - Data c3
```

Illustration 41: Server message

Distribution	Ubuntu 14.04 LTS
Memory	2.0 GB
Processor	Intel Core i3-3220 CPU @ 3.30 GHz
OS type	64-bit
Graphics	Gallium 0.4 on llvmpipe (LLVM 3.5, 128 bits)
Gnome	Gnome-shell not installed
Virtualization	Oracle (Virtualbox 4.3.20 r96997)
C compiler	Gcc 4.8.2
Python	2.7.6

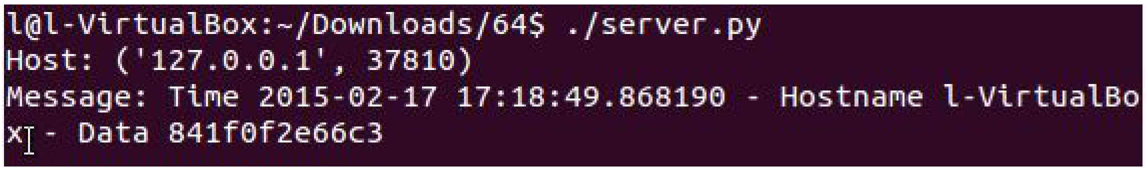
Table 9: Ubuntu 64-bit

For the 64-bit version of Ubuntu, executing the code of the buffer overflow [31] we present the screenshots of the successful detection of a buffer overflow with the corresponding message received by the system administrator:



```
l@l-VirtualBox:~/Downloads/64$ ./64full vul
0x7fff49724d70
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA
Error! Sends me to=0x4141414141414141 Current RIP=0x400600
INTERNAL: Clicks: 11937618 (11.937618 seconds)
```

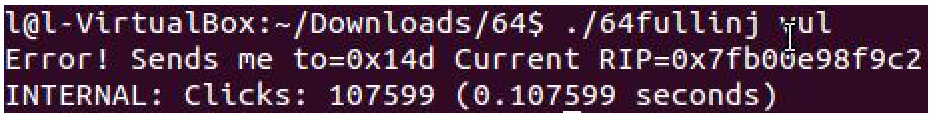
Illustration 42: Buffer overflow



```
l@l-VirtualBox:~/Downloads/64$ ./server.py
Host: ('127.0.0.1', 37810)
Message: Time 2015-02-17 17:18:49.868190 - Hostname l-VirtualBo
x - Data 841f0f2e66c3
```

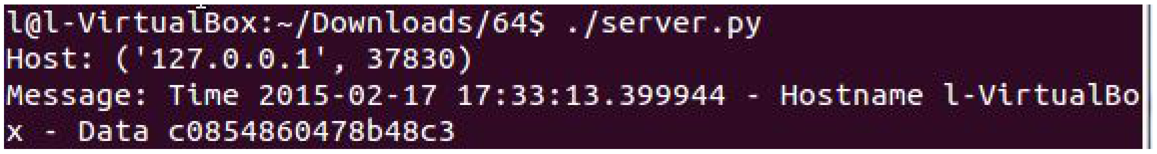
Illustration 43: Server message

For the 64-bit version of Ubuntu, executing the code for the injection (Section 8.2), we present the screenshots of the successful detection of a code injection with the corresponding message received by the system administrator:



```
l@l-VirtualBox:~/Downloads/64$ ./64fullinj vul
Error! Sends me to=0x14d Current RIP=0x7fb00e98f9c2
INTERNAL: Clicks: 107599 (0.107599 seconds)
```

Illustration 44: Injection



```
l@l-VirtualBox:~/Downloads/64$ ./server.py
Host: ('127.0.0.1', 37830)
Message: Time 2015-02-17 17:33:13.399944 - Hostname l-VirtualBo
x - Data c0854860478b48c3
```

Illustration 45: Server message

5.4 Time

We now present the results of our benchmarking. We do not include the injection version because we manually inject code and so it does not represent any significant data.

Operating System	Executable	Time
OpenSUSE	64full with loop	0.23 seconds
	64full with vul	0.25 seconds
	64full with /usr/bin/cal	2.23 seconds
	64fast with loop	0.03 seconds
	64fast with vul	0.002 seconds
	64fast with /usr/bin/cal	0.01 seconds
	32full with loo	0.33 seconds
	32full with vuln	0.27 seconds
	32full with /usr/bin/cal	2.84 seconds
Fedora	64full with loop	0.28 seconds
	64full with vul	0.25 seconds
	64full with /usr/bin/cal	1.53 seconds
	64fast with loop	0.02 seconds
	64fast with vul	0.003 seconds
	64fast with /usr/bin/cal	0.008 seconds
	32full with loo	0.41 seconds
	32full with vuln	0.47 seconds
	32full with /usr/bin/cal	1.33 seconds
Ubuntu	64full with loop	0.46 seconds
	64full with vul	0.37 seconds
	64full with /usr/bin/cal	1.58 seconds
	64fast with loop	0.02 seconds
	64fast with vul	0.001 seconds
	64fast with /usr/bin/cal	0.007 seconds
	32full with loo	0.52 seconds
	32full with vuln	0.21 seconds
	32full with /usr/bin/cal	1.08 seonds

Table 10: Benchmarking

Based on the results shown above we can conclude that the full version for both the 64-bit and 32-bit version require more time to analyze a process, with the time increasing proportionately to the size or number of operations executed by the target process. For example it took somewhere between 1.08 and 2.84 seconds to successfully analyze and execute /usr/bin/cal in 32-bit systems, which is a simple calendar application. Meanwhile the time it takes to analyze the same application on 64-bit systems varies from distribution to distribution with OpenSUSE taking the most time (2.23 seconds) and Fedora taking the least (1.53 seconds). However, as previously stated, these were virtual machines so any conclusion that would attempt to point out key differences between distributions could be overshadowed by external variables such as the application used to virtualize, as well as the hardware of the host system and its ability to perform virtualization. On the other hand, the fast version performed relatively uniformly across all distributions with a decrease in the time needed to perform its analysis. For example we went from 2.23 seconds with the full version to 0.01 seconds with the fast version on the OpenSUSE 64-bit distribution. The other two distributions follow the same pattern of reducing the time needed and therefore successfully supporting our original argument that the fast version performs the task while reducing the time overhead.

6 Conclusions

At the beginning of this thesis we established the following objectives:

1) Develop a countermeasure against stack pointer manipulation in the GNU/Linux operating systems.

2) Automate the detection process at execution time.

3) Notify the system administrator in real time.

4) Minimize the consumption of system resources.

Our solution is a working countermeasure against stack pointer manipulation in GNU/Linux operating systems as shown in the Section 5. It automatically detects anomalies during run-time and if a problem is detected then a message is automatically sent to the system administrator over the network, as explained in Section 4. We also developed a faster version of our solution which minimizes the consumption of resources (mainly, time). We therefore conclude that our work is successful and meets our desired requirements of automation and notification.

In comparison with the previous work, specifically Benjamin Teissier's work, we fix the problem of the size of the text segment and we also work in user space and so we completely nullify the problems associated to the sensitivity of kernel space and its limited size. We also completely rework our approach in order to intercept opcodes in real time, during the execution of each and every instruction, while his work patches the process at launch time instead of intercepting the instructions. One of the main issues of his work was the architecture (32-bit only). By contrast we considered 64-bit as well as 32-bit architectures. Finally, the time issue still remains to an extent. While it could be argued that our approach also takes up a relatively large amount of time, we also attempted to solve that issue with our fast version. This limitation will be elaborated on in the next subsection.

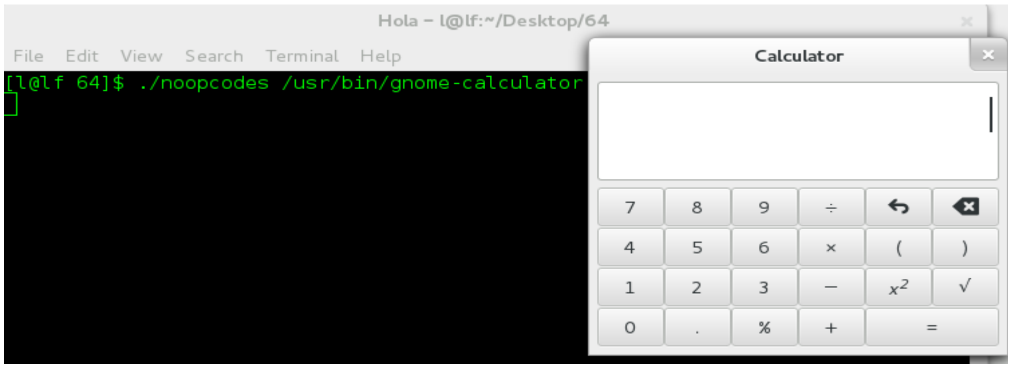
We have successfully applied our solution to various command-line programs. However, we encountered problems while handling programs using graphical user interfaces (see Section 6.1 below). In general, we conclude that our work is a fully functional applicable result that would be better suited to server environments where graphical user interfaces are not widely used. Needless to say, these are the most sensitive and critical assets of a network and companies spend thousands of dollars in infrastructure and software to protect them. On top of that we completely operate at execution time which makes our solution applicable even in circumstances that prevent other preventive measures against stack pointer modifications such as canaries to operate. In particular, closed source applications as well as legacy applications where we do not have access to the source code are equally protected by our solution.

In comparison with other related work as presented in Chapter 3 we note that some of the previously proposed measures require the addition of extra data or even an entire re-engineering of certain mechanisms. By contrast our work operates natively under any GNU/Linux distribution without the need to add data to addresses like canaries, or randomize the entire memory structure of processes like ASLR, or add support to hardware. In comparison, we operate as just another user space application without the need to modify the kernel, libraries, hardware or even the actual target process. In essence, we successfully analyze processes without modifying anything other than whatever normal system resources we require, just like any other application.

6.1 Limitations

The limitations that our work presents hover mostly around the time issue. As explained before there are not many things we can do regarding ptrace. However the bigger the process being executed is, the bigger the supplementary buffer becomes, and thus the longer it will take to perform the operations. However, we also developed a faster version of our solution which attempts to work around the time problem and it does precisely that at least for specific scenarios. The problem with our fast version arises from the fact that the theory behind it assumes certain things, as well as takes risks with its mechanics. The issue becomes evident when we take a look at what we are intercepting that is, calls and rets. With our normal version, every single call and ret will be accounted for and monitored; however with our fast version, only those calls or rets that happen when a write system call is detected will be accounted for. This leads to the first potential problem: if a call is issued during execution-time outside of a write system call, and a write system call takes place followed by a ret that will attempt to go back to said call, the fast version of our solution will issue a false positive. Since the call was not accounted for due to it taking place outside of the write system call, by the time the ret is issued and evaluated, the buffer will not have the proper value stored and the fast version of our solution will never match the current value in the stack to any value in the buffer even though it is a legit return.

Another problem possibly related to the time issue is when dealing with graphical user interfaces. The expected behavior is that even if it takes a long time to evaluate each instruction, eventually the graphical user interface will be launched for a given process. In practice there are instances when this interface does not launch. This is a problem that requires a high level of expertise to analyze and figure out, down to the assembly code. As an example of an instance when this occurs we consider Gnome-based graphical user interfaces and more precisely gnome-calculator, normally located at /usr/bin/gnome-calculator. Even if there is an instance of gnome-calculator running on the system, and therefore we assume that certain libraries have been loaded, our solution still "hangs" forever due to, we assume, a timeout issue. This is supported by the fact that when we run the code without checking for opcodes at all, the graphical user interfaces launch without a problem. We can visualize it with the following execution of gnome-calculator:



```
[l@lf 64]$ ./noopcodes /usr/bin/gnome-calculator
```

Illustration 46: No opcodes

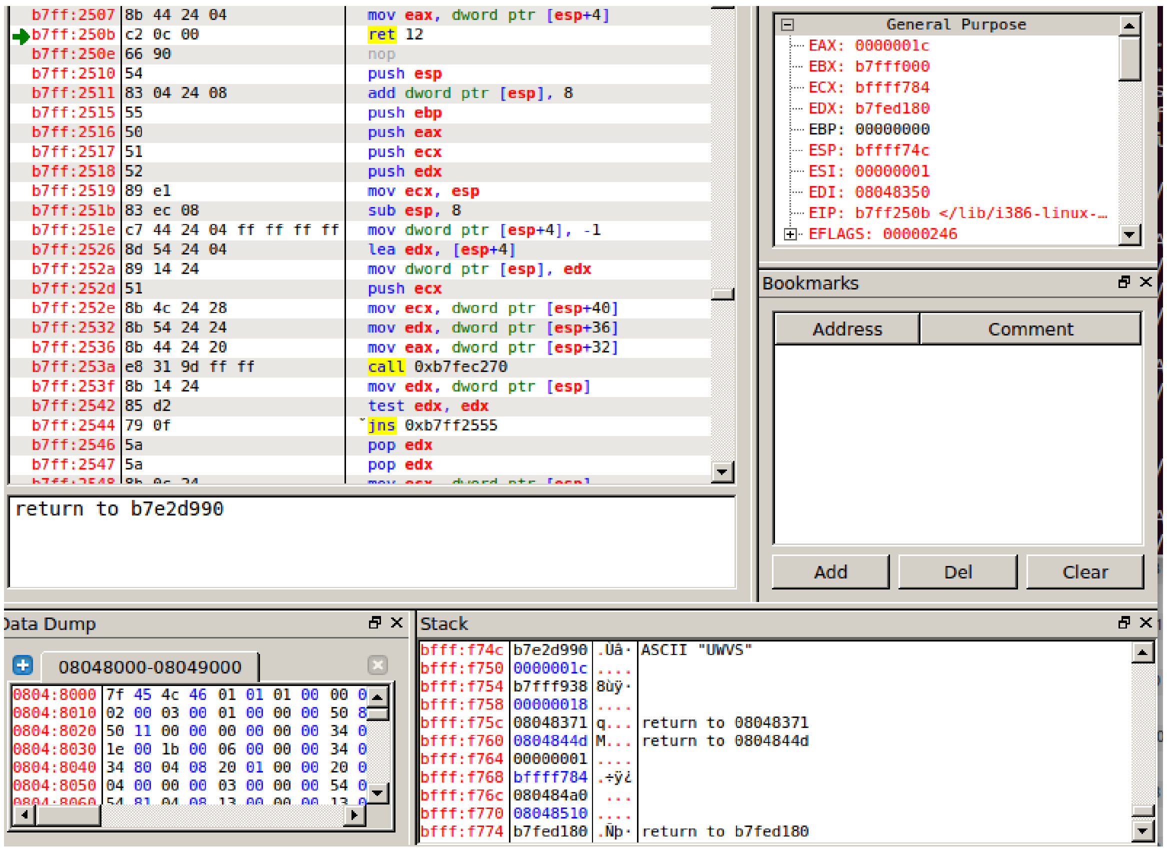
The problem disappears once we eliminate graphical user interfaces and run command-line executables. A possible lead towards a solution to this issue might be in the way threads are generated within the target process. We were however not able to confirm that the multi-threaded nature of GUI programs is the cause of the above issue. Indeed, we ran our solution on simple (that is, non-GUI) programs that use fork(), clone(), and pthread_create(). In all cases the programs ran successfully under our monitoring system. A solution to the GUI problem involves other, more complex topics such as operating system scheduling algorithms that fall outside the scope of our work. This being said, we believe that our solution is still fully applicable to production servers, which normally do not run graphical user interfaces at all.

The last issue is the fact that the fast version does not work on 32-bit operating systems. Once again, this problem requires a high level of expertise to solve, down to assembly code.

In conclusion it is necessary to state that the fast version is nothing more than an educated attempt at fixing the time issue compared with our fully-working, normal version. Thus, while we are aware that countless arguments could be made against this fast version, we believe that our work on the matter represents a first step towards a fix, so we present the fast version as an added bonus to our fully-working, normal version.

6.1.1 Buffer optimization

There is a probable way to maximize the efficiency of our buffer. As previously mentioned, in the 32-bit version there exists the problem of the opcode c2 for ret due to values getting pushed on the stack. In this case, we can verify that when the ret is issued, the address to which it returns is not preceded by a call, which is what we would expect and actively intercept. This is why we force our "if" condition for ret to only intercept c2 if it is followed by any two values:



```
b7ff:2507 8b 44 24 04              mov eax, dword ptr [esp+4]
b7ff:250b c2 0c 00                 ret 12
b7ff:250e 66 90                    nop
b7ff:2510 54                       push esp
b7ff:2511 83 04 24 08              add dword ptr [esp], 8
b7ff:2515 55                       push ebp
b7ff:2516 50                       push eax
b7ff:2517 51                       push ecx
b7ff:2518 52                       push edx
b7ff:2519 89 e1                    mov ecx, esp
b7ff:251b 83 ec 08                 sub esp, 8
b7ff:251e c7 44 24 04 ff ff ff ff  mov dword ptr [esp+4], -1
b7ff:2526 8d 54 24 04              lea edx, [esp+4]
b7ff:252a 89 14 24                 mov dword ptr [esp], edx
b7ff:252d 51                       push ecx
b7ff:252e 8b 4c 24 28              mov ecx, dword ptr [esp+40]
b7ff:2532 8b 54 24 24              mov edx, dword ptr [esp+36]
b7ff:2536 8b 44 24 20              mov eax, dword ptr [esp+32]
b7ff:253a e8 31 9d ff ff           call 0xb7fec270
b7ff:253f 8b 14 24                 mov edx, dword ptr [esp]
b7ff:2542 85 d2                    test edx, edx
b7ff:2544 79 0f                   ˇjns 0xb7ff2555
b7ff:2546 5a                       pop edx
b7ff:2547 5a                       pop edx
```

```
return to b7e2d990
```

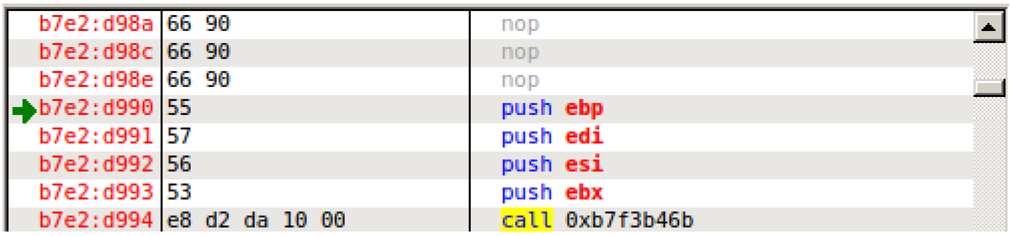
```
General Purpose
EAX: 0000001c
EBX: b7fff000
ECX: bffff784
EDX: b7fed180
EBP: 00000000
ESP: bffff74c
ESI: 00000001
EDI: 08048350
EIP: b7ff250b </lib/i386-linux-…
EFLAGS: 00000246
```

```
08048000-08049000
0804:8000 7f 45 4c 46 01 01 01 00 00 0
0804:8010 02 00 03 00 01 00 00 00 50 8
0804:8020 50 11 00 00 00 00 00 00 34 0
0804:8030 1e 00 1b 00 06 00 00 00 34 0
0804:8040 34 80 04 08 20 01 00 00 20 0
0804:8050 04 00 00 00 03 00 00 00 54 0
```

```
bfff:f74c b7e2d990 .Ùâ· ASCII "UWVS"
bfff:f750 0000001c ....
bfff:f754 b7fff938 8ùÿ·
bfff:f758 00000018 ....
bfff:f75c 08048371 q... return to 08048371
bfff:f760 0804844d M... return to 0804844d
bfff:f764 00000001 ....
bfff:f768 bffff784 .÷ÿ¿
bfff:f76c 080484a0  ...
bfff:f770 08048510 ....
bfff:f774 b7fed180 .Ñþ· return to b7fed180
```

Illustration 47: Ret and stack viewer

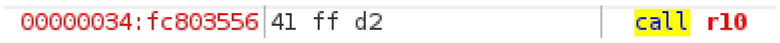
This example above is under 32-bit Ubuntu and we can see that the stack contains the ASCII symbols “UWVS” and it will return to the address b7e2d990 where based on our assumptions we are expecting a call. However after execution of the ret, we can verify that there is no associated call at that location:



```
b7e2:d98a 66 90          nop
b7e2:d98c 66 90          nop
b7e2:d98e 66 90          nop
b7e2:d990 55             push ebp
b7e2:d991 57             push edi
b7e2:d992 56             push esi
b7e2:d993 53             push ebx
b7e2:d994 e8 d2 da 10 00 call 0xb7f3b46b
```

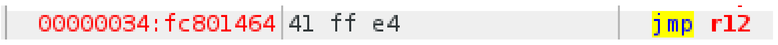
Illustration 48: Return

A similar issue happens with our 64-bit versions. However in this case, the issue is with the opcode for call. Indeed, one of the opcodes for call is rex.w + ff; however the rex.w prefix only happens in long-mode which in turn occurs during a series of possible conditions [32]. Since our code has time issues as it is, including a bigger list of conditions to look for during execution-time would not be optimal. Our solution to this issue was to listen to all instructions that met the criteria of ff values preceded by any two values. This in turn simplifies the problem and allows us to intercept both jmp and call instructions:



```
00000034:fc803556 41 ff d2     call r10
```

Illustration 49: Call



```
00000034:fc801464 41 ff e4     jmp r12
```

Illustration 50: Jmp

It is theoretically possible to optimize the buffer for both versions if simpler solutions to the opcode c2 problem for the 32-bit version and to the rex.w prefix for 64-bit version are developed.

6.2 Future work

Several issues remain with our solution, as outlined in the previous section. Solving these issues is one of our continuing interests on the matter.

Due to the nature of our solution, it is possible to port it over to other operating systems, mainly, BSD-based operating systems (such as NetBSD and OpenBSD), but potentially all Unix-based operating systems. The mechanisms we utilize for our solution are similar between all such operating systems, so the porting process is simple. However the syntax does change and some other minor details need to be worked out as well. Another operating system that could be used to port is Mac OS X. Since OS X is roughly based off BSD, we can still find ptrace in there. However the functionality of ptrace under OS X is greatly reduced; nonetheless we believe that this could potentially be worked around. Another major, alternate operating system is Solaris. Since it is based of Unix, we can also find ptrace built into it. In short, wherever we can find ptrace, we can port our solution, assuming of course that the core functionality is still available.

7 Bibliography

[1] J. Pinto, *Robotics Technology Trends: The Future of Robots*, http://www.automation.com/library/articles-white-papers/articles-by-jim-pinto/robotics-technology-trends, retrieved Feb. 2015.

[2] B. Boyce, *A history of viruses on Linux*, (Nov. 27 2010), http://www.neowin.net/news/a-history-of-viruses-on-linux, retrieved Feb. 2015.

[3] C. Janssen, *What is Assembly Language?*, http://www.techopedia.com/definition/3903/assembly-language, retrieved Feb. 2015.

[4] C. Janssen, *What is a Memory address?*, http://www.techopedia.com/definition/323/memory-address, retrieved Feb. 2015.

[5] C. Janssen, *What is x86 Architecture?*, http://www.techopedia.com/definition/5334/x86-architecture, retrieved Feb. 2015.

[6] Computer Hope, *What is the difference between 32-bit and 64-bit CPU?*, http://www.computerhope.com/issues/ch001498.htm, retrieved Feb. 2015.

[7] Tutorialspoint, *Assembly – Registers*, http://www.tutorialspoint.com/assembly_programming/assembly_registers.htm, retrieved Feb. 2015.

[8] Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A, 2B, and 2C: Instruction Set Reference, A-Z, http://download.intel.com/products/processor/manual/325383.pdf retrieved Feb. 2015.

[9] University of Maryland's Computer Science department, *Understanding the stack*, Class notes (Summer 2003) http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html, retrieved Feb. 2015.

[10] R. Stallman, *Linux and the GNU System*, https://www.gnu.org/gnu/linux-and-gnu.html, retrieved Feb. 2015.

[11] How-To Geek, *10 of the Most Popular Linux Distributions Compared*,

http://www.howtogeek.com/191207/10-of-the-most-popular-linux-distributions-compared/?PageSpeed=noscript, retrieved Feb. 2015.

[12] M. Ben-Yehuda, *10 Things Every Linux Programmer Should Know, Linux Misconceptions in 30 Minutes*, presentation at the Linux Kernel Workshop (Mar. 2004), IBM Haifa Research Labs, http://www.mulix.org/lectures/kernel_workshop_mar_2004/things.pdf.

[13] ptrace(2) – Linux man page, http://linux.die.net/man/2/ptrace, retrieved Feb. 2015.

[14] Aleph One, *Smashing the Stack for Fun and Profit*, Phrack 7:49 (1996) http://insecure.org/stf/smashstack.html.

[15] OWASP, *Buffer Overflow*, https://www.owasp.org/index.php/Buffer_Overflow, retrieved Feb. 2015.

[16] OWASP, *Code Injection*, https://www.owasp.org/index.php/Code_Injection, retrieved Feb. 2015.

[17] Python Software Fundation, *Python for Begginers*, https://www.python.org/about/gettingstarted/, retrieved Feb. 2015.

[18] C. Janssen. *What is C (Programming Language)?*, http://www.techopedia.com/definition/24068/c-programming-language-c, retrieved Feb. 2015.

[19] Address Space Layout Randomization, (Mar. 2003), https://pax.grsecurity.net/docs/aslr.txt, retrieved Feb. 2015.

[20] G. Duarte. *Epilogues, Canaries, and Buffer Overflows*, (Mar. 19 2014), http://duartes.org/gustavo/blog/post/epilogues-canaries-buffer-overflows/, retrieved Feb. 2015.

[21] J. Deckard, *Defeating Overflow Attacks*, GSEC Practical Assignment, Version 1.4b Option 1, SANS Institute InfoSec Reading Room (April. 14, 2004), http://www.sans.org/reading-room/whitepapers/securecode/defeating-overflow-attacks-1403.

[22] P. Silberman and R. Johnson, *A Comparison of Buffer Overflow Prevention Implementations and Weaknesses*, presentation at Black Hat USA, Caesar's Palace, Las Vegas, NV, USA (Jul. 2004) http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf.

[23] W. Chuang, S. Narayanasamy, B. Calder and R. Jhala, *Bounds Checking with Taint-Based Analysis*, in Proceedings of the 2nd International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'07), Ghent, Belgium (Jan. 2007), pp 71-86 https://cseweb.ucsd.edu/~wchuang/HiPEAC-07-TaintBounds.pdf.

[24] K. Piromsopa and R. J. Enbody, *Secure Bit2: Transparent, Hardware Buffer-Overflow Protection*, Michigan State University, http://www.cse.msu.edu/~cse825/SBit2.pdf, retrieved Feb. 2015.

[25] M. Dalton, H. Kannan and C. Kozyrakis, *Real-World Buffer Overflow Protection for Userspace & Kernelspace*, in Proceedins of the 17th USENIX Security Symposium (USENIX Security '08), San Jose,

California, USA (Jul. 2008) https://www.usenix.org/legacy/event/sec08/tech/full_papers/dalton/dalton_html/.

[26] M. Zhivich, T. Leek and R. Lippmann, *Dynamic Buffer Overflow Detection*, in Proceedings of the 2005 Workshop on the Evaluation of Software Defect Detection Tools, Chicago, IL, USA (June 12, 2005) https://www.cs.umd.edu/~pugh/BugWorkshop05/papers/61-zhivich.pdf.

[27] *Non-Executable Pages Design and Implementation*, (May 2003), http://pax.grsecurity.net/docs/noexec.txt, retrieved Feb. 2015

[28] Z. Shao, C. Xue, Q. Zhuge, E. H.-M. Sha, *Security Protection and Checking in Embedded System Integration Against Buffer Overflow Attacks*, in Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2004), Volume I, Apr. 2004, pp. 409-413.

[29] Z. Shao, J. Cao, K. C. C. Chan, C. Xue, E. H.-M.Sha, *Hardware/Software Optimization for Array & Pointer Boundary Checking Against Buffer Overflow Attacks*, Journal of Parallel and Distributed Computing, 66:9, 2006, pp. 1129-1136.

[30] B. Teissier, *An Approach to Stack Overflow Counter-Measures Using Kernel Properties*, M.Sc. Thesis, Bishop's University, Sherbrooke, Quebec, Canada, Nov. 2013.

[31] Mr. Un1k0d3r RingZer0 Team, *64 Bits Linux Stack Buffer Overflow*, http://www.exploit-db.com/wp-content/themes/exploit/docs/33698.pdf, retrieved Feb. 2015.

[32] x86-64 Instruction Encoding, http://wiki.osdev.org/X86-64_Instruction_Encoding#REX_prefix, retrieved Feb. 2015.

[33] P. Roberts, *At the Vulnerability Oscars, The Winner Is... Buffer Overflow!!*, https://www.veracode.com/blog/2013/02/at-the-vulnerability-oscars-the-winner-is-buffer-overflow, retrieved April 2015.

8 Appendices

8.1 64fast.c

To enable injection, notice the commented line of of code that mentions injection

```
//Erick Leon, 2015
//Based off the code by Pradeep Padala
//    http://www.linuxjournal.com/article/6100?page=0,2
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/user.h>
#include <sys/reg.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stdint.h>
#include <sys/syscall.h>
#include <signal.h>
#include <unistd.h>
#include <time.h>

int main(int argc, char** argv){//main function
clock_t t;//benchmarking
t=clock();//benchmarking
int calll=0;//updates destiny after call
int i;//for
int counter=0;
FILE *ptr_file; //file storing hex values
uint64_t instr,check,dest;//store the hex values of instructions and eip/rip
uint64_t bfr[900000];
pid_t child; //child process
const int long_size = sizeof(long);
  if(argc < 2){
  fprintf(stderr, "Invalid\n");
  return -1;
  }//if argc
child = fork();//fork the child from the parent
  if(child == 0){ //no child process exist
  ptrace(PTRACE_TRACEME, 0, NULL, NULL); //trace the process
  execl(argv[1], argv[1] , (char*)0); //run the program
  }else{
  int status; //variables to control status
    union u{
    long val;
    char chars[long_size];
    }data;
  struct user_regs_struct regs; //regs for EIP or RIP
  int start = 0;
    while(1){
    wait(&status);//stores the status of wait
      if(WIFEXITED(status))//if done
      break;
    ptrace(PTRACE_GETREGS,child, NULL, &regs);//get regs
      if(start == 1){
      instr = ptrace(PTRACE_PEEKTEXT,child, regs.rip,NULL);//obtain text
    if (calll == 1){//if a call has happened, update destiny
     dest = ptrace(PTRACE_PEEKDATA, child, regs.rsp, 0);//next instruction after
call
    bfr[counter]=dest;//store the destination in a buffer
    counter++;//increase to the next position
    }
    calll=0;//reset the call counter, i.e. call has NOT happened
/*Create a file to store the hex values of instr*/
  ptr_file =fopen("o", "w"); //open file to write
  fprintf(ptr_file,"%lx\n", instr);//write hex values to file
  fclose(ptr_file);//close file
  char ch[16];//stores single characters of hex values
  int n = 0;//counter
  ptr_file =fopen ("o","r");//open file to read
    while (!feof(ptr_file)){//while no ending of file
    ch[n] = fgetc(ptr_file);//get individual characters
    n++;//increase counter
    }//while
  fclose(ptr_file);//close file
/*End of file creation*/
    if ((ch[n-4]=='e'&&ch[n-3]=='8')||(ch[n-4]=='9'&&ch[n-3]=='a')||
        (ch[n-4]=='f'&&ch[n-3]=='f')||(ch[n-6]=='f'&&ch[n-5]=='f')){
                                                              //if opcodes for call
    calll=1;//a call has happened
    //printf("Call at RIP: 0x%lx instruction: 0x%lx\n",regs.rip,instr);
    }//if opcodes for call
    if((ch[n-4]=='c'&&ch[n-3]=='3')||(ch[n-4]=='c'&&ch[n-3]=='b')||
       (ch[n-4]=='c'&&ch[n-3]=='2')||(ch[n-4]=='c'&&ch[n-3]=='a')){
                                                               //if opcodes for ret
    counter--;//decrease position on array
    //ptrace(PTRACE_POKEDATA, child,regs.rsp, 333); //Code injection
    check = ptrace(PTRACE_PEEKDATA, child, regs.rsp, 0); //val in current rsp stack
```

```
      //printf("Ret  Going  to:  0x%lx   Supposed  to  go  to:  0x%lx  instruction:  0x
%lx\n",check, dest, instr);
    int ii = 0;//if ret has happened
      for(i=counter;i>=0;i--){//check all stored dest
        if(check == bfr[i]){//if next position from call is different from where
                            // the rsp stack wants to send me
        ii=1;//if a ret was fine to go
        break;
        }//if check
      }//for i=counter
      if(ii!=1){//if ret was NOT fine to go
             printf("Error!  Sends  me  to=0x%lx  Current  RIP=0x%lx  instr:  0x
%lx\n",check,regs.rip,instr);//Shows the problem
      kill(child,SIGKILL); //kill the process
      system("./client.py");//Notify the sysadmin
      t=clock()-t;//benchmarking
                       printf("INTERNAL:    Clicks:    %d    (%f    seconds)\n",t,
((float)t)/CLOCKS_PER_SEC);//benchmarking
      exit(0);//end
      }//if ii
    }//if opcodes for ret
  //wait(&status);//Wait for next stop
      }//if start = 1
   if((regs.orig_rax == SYS_write)){//every new instance
   start = 1;//initialize
   ptrace(PTRACE_SINGLESTEP, child,NULL, NULL);//singlestep the child
   }else ptrace(PTRACE_SYSCALL, child, NULL, NULL);//begin listening to syscalls
 }//while 1
}//if-else
t=clock()-t;//benchmarking
printf("EXTERNAL: Clicks: %d (%f seconds)\n",
        t,((float)t)/CLOCKS_PER_SEC);//benchmarking
return 0;
}//main
```

8.2 64full.c

To enable injection, notice the commented line of of code that mentions injection.

```
//Erick Leon, 2015
//Based off the code by Eli Bendersky
//    http://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1.html
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/reg.h>
#include <sys/user.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <signal.h>
```

```
#include <syscall.h>
#include <stdint.h>
#include <sys/ptrace.h>
#include <time.h>

void name(const char* programname){
  if(ptrace(PTRACE_TRACEME, 0, 0, 0) < 0){//Trace the program
  perror("ptrace");
  return;
  }
execl(programname, programname, (char *)0);//Name of the program to trace
}

void analyze(pid_t child_pid){
clock_t t;//benchmarking
t=clock();//benchmarking
int calll=0;//updates destiny after call
int i;//for
int counter=0; //counter to manipulate our buffer
FILE *ptr_file; //file storing hex values
uint64_t instr,check,dest,op,op2;//store the hex values of instructions and eip/rip
and opcodes
uint64_t bfr[900000]; //our buffer
int wait_status;
wait(&wait_status);//wait for child process to stop at first instruction
  while(WIFSTOPPED(wait_status)){
  struct user_regs_struct regs;//eip or rip
  ptrace(PTRACE_GETREGS, child_pid, 0, &regs);//obtain regs eip or rip
  instr = ptrace(PTRACE_PEEKTEXT, child_pid, regs.rip, 0);//instr. at current rip
    if (calll == 1){//if a call has happened, update destiny
    dest = ptrace(PTRACE_PEEKDATA, child_pid, regs.rsp, 0);//next instr. after call
    bfr[counter]=dest;//store the destination in a buffer
    counter++;//increase to the next position
    }//end of if call has happened
  calll=0;//reset the call counter, i.e. call has NOT happened
  op = instr & 0xff; //opcode for first byte
  op2 = instr & 0xff00; //opcode for second byte, rex.w prefix
    if (op==0xe8 || op==0x9a || op==0xff || op2 == 0xff00){//if opcodes for call
    calll=1;//a call has happened
    //printf("Call at RIP: 0x%lx instruction: 0x%lx\n",regs.rip, instr);
    }//if opcodes for call
    if (op==0xc3 || op==0xcb || op==0xc2 || op==0xca){//if opcodes for ret
    counter--;//decrease position on array
    //ptrace(PTRACE_POKEDATA, child_pid,regs.rsp, 333); //Code injection
    check = ptrace(PTRACE_PEEKDATA, child_pid, regs.rsp, 0);//val in curr rsp stack
    //printf("Ret Going to: 0x%lx  Supposed to go to: 0x%lx instruction: 0x
%lx\n",check, dest, instr);
    int ii = 0;//if ret has happened
      for(i=counter;i>=0;i--){//check all stored dest
        if(check == bfr[i]){//if next position from call is different from where
                            // the rsp stack wants to send me
        ii=1;//if a ret was fine to go
        break;
        }//if check
      }//for i=counter
      if(ii!=1){//if ret was NOT fine to go
      printf("Error! Sends me to=0x%lx Current RIP=0x%lx Instr=0x
%lx\n",check,regs.rip,instr);//Shows the problem
      kill(child_pid,SIGKILL); //kill the process
      //Create a file to store the hex values of instr
      ptr_file =fopen("o", "w"); //open file to write
      fprintf(ptr_file,"%lx\n", instr);//write hex values to file
      fclose(ptr_file);//close file
      //End of file creation
      system("./client.py");//Notify the sysadmin
      t=clock()-t;//benchmarking
      printf("INTERNAL: Clicks: %d (%f seconds)\n",t,
((float)t)/CLOCKS_PER_SEC);//benchmarking
      exit(0);//end
      }//if ii
    }//if opcodes for ret
    if(ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0) < 0){//Single step it
    perror("ptrace");
    return;
    }//if singlestep
  wait(&wait_status);//Wait for next stop
  }//while general
}//function

int main(int argc, char** argv){//main
clock_t t;//benchmarking
t=clock();//benchmarking
pid_t child_pid;//child process
  if(argc < 2){
  fprintf(stderr, "Invalid\n");
  return -1;
  }//if argc
child_pid = fork();
  if(child_pid == 0)
  name(argv[1]);
  else if(child_pid > 0)
  analyze(child_pid);
  else{
  perror("fork");
  return -1;
  }//if-else
t=clock()-t;//benchmarking
printf("EXTERNAL: Clicks: %d (%f seconds)\n",t,
((float)t)/CLOCKS_PER_SEC);//benchmarking
return 0;
}//main
```

8.3 32full.c

To enable injection, notice the commented line of of code that mentions injection.

```
//Erick Leon, 2015
//Based off the code by Eli Bendersky
//    http://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1.html
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/reg.h>
#include <sys/user.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <signal.h>
#include <syscall.h>
#include <stdint.h>
#include <sys/ptrace.h>
#include <time.h>

void name(const char* programname){
  if(ptrace(PTRACE_TRACEME, 0, 0, 0) < 0){//Trace the program
  perror("ptrace");
  return;
  }
execl(programname, programname, (char *)0);//Name of the program to trace
}

void analyze(pid_t child_pid){
clock_t t;//benchmarking
t=clock();//benchmarking
int calll=0;//updates destiny after call
int i;//for
int counter=0;
FILE *ptr_file; //file storing hex values
long instr,check,dest,op,op2;//store the hex values of instructions and eip/eip
long bfr[900000];
int wait_status;
wait(&wait_status);//wait for child process to stop at first instruction
  while(WIFSTOPPED(wait_status)){
  struct user_regs_struct regs;//eip or eip
  ptrace(PTRACE_GETREGS, child_pid, 0, &regs);//obtain regs eip or eip
  instr = ptrace(PTRACE_PEEKTEXT, child_pid, regs.eip, 0);//instr. at current eip
    if (calll == 1){//if a call has happened, update destiny
    dest = ptrace(PTRACE_PEEKDATA, child_pid, regs.esp, 0);//next instr. after call
    bfr[counter]=dest;//store the destination in a buffer
    counter++;//increase to the next position
    }
  calll=0;//reset the call counter, i.e. call has NOT happened
  op = instr & 0xff; //opcode for first byte
  op2 = instr & 0xff00; //opcode for second byte, c2 problem
    if (op==0xe8 || op==0x9a || op==0xff || op2==0xff00){//if opcodes for call
    calll=1;//a call has happened
    //printf("Call at eip: 0x%lx instr; 0x%lx\n",regs.eip, instr);
    }//if opcodes for call
    if (op==0xc3 || op==0xcb || op==0xca || op2 ==0x00c2){//if opcodes for ret
    counter--;//decrease position on array
    //ptrace(PTRACE_POKEDATA, child_pid,regs.esp, 333); //Code injection
    check = ptrace(PTRACE_PEEKDATA, child_pid, regs.esp, 0);//val in curr esp stack
    //printf("Ret Going to: 0x%lx  Supposed to go to: 0x%lx\n",check, dest);
    int ii = 0;//if ret has happened
      for(i=counter;i>=0;i--){//check all stored dest
        if(check == bfr[i]){//if next position from call is different from where
                            // the esp stack wants to send me
        ii=1;//if a ret was fine to go
        break;
        }//if check
      }//for i=counter
      if(ii!=1){//if ret was NOT fine to go
      printf("Error! Sends me to=0x%lx Current eip=0x%lx  instr: 0x
%lx\n",check,regs.eip,instr);//Shows the problem
      kill(child_pid,SIGKILL); //kill the process
      /*Create a file to store the hex values of instr*/
      ptr_file =fopen("o", "w"); //open file to write
      fprintf(ptr_file,"%lx\n", instr);//write hex values to file
      fclose(ptr_file);//close file
      /*End of file creation*/
      system("./client.py");//Notify the sysadmin
      t=clock()-t;//benchmarking
      printf("INTERNAL: Clicks: %d (%f seconds)\n",t,
((float)t)/CLOCKS_PER_SEC);//benchmarking
      exit(0);//end
      }//if ii
    }//if opcodes for ret
    if(ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0) < 0){//Single step it
    perror("ptrace");
    return;
    }//if singlestep
  wait(&wait_status);//Wait for next stop
  }//while general
}//function

int main(int argc, char** argv){//main
clock_t t;//benchmarking
t=clock();//benchmarking
pid_t child_pid;//child process
  if(argc < 2){
  fprintf(stderr, "Invalid\n");
  return -1;
  }//if argc
child_pid = fork();
  if(child_pid == 0)
  name(argv[1]);
  else if(child_pid > 0)
  analyze(child_pid);
  else{
  perror("fork");
  return -1;
  }//if-else
t=clock()-t;//benchmarking
printf("EXTERNAL: Clicks: %d (%f seconds)\n",t,
((float)t)/CLOCKS_PER_SEC);//benchmarking
return 0;
}//main
```

8.4 loop.c

```
#include <stdio.h>
#include <stdlib.h>
int main()
{   int i;
    for(i = 0;i < 10; ++i) {
        printf("Testing : %d\n", i);
        sleep(2);
    }
    return 0;
}
```

8.5 vul.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char **argv) {
char buffer[256];
printf("%p\n", buffer);
strcpy(buffer,
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA");
printf("%s\n", buffer);
return 0;
}
```

8.6 client.py

```
#!/usr/bin/env python
import socket
import datetime
host = '127.0.0.1'
port = 5555
```

```
f=open("o","r+")
str=f.read(16);
dat=str
f.close()
msg = 'Time %s - Hostname %s - Data %s\n'%
(datetime.datetime.now(),socket.gethostname(),dat)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
s.send(msg)
s.close()
```

8.7 server.py

```
#!/usr/bin/env python
import socket
host = '127.0.0.1'
port = 5555
bfr = 2000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
s.listen(1)
conn, addr = s.accept()
print 'Host:', addr
while 1:
  data = conn.recv(bfr)
  if not data:
    break
  elif data == 'kill':
    conn.close()
    sys.exit()
  else:
    print "Message:", data
```