Defeating Memory Error Exploits by Program Diversification and Process Replication

Marc Nimmerrichter

Technical Report
RHUL–MA–2014–3
01 September 2014

Information Security Group
Royal Holloway, University of London
Egham, Surrey TW20 0EX,
United Kingdom

www.ma.rhul.ac.uk/tech
Defeating Memory Error Exploits by Program Diversification and Process Replication

Supervisor: Dr. Lorenzo Cavallaro

Submitted as part of the requirements for the award of the MSc in Information Security at Royal Holloway, University of London

I declare that this assignment is all my own work and that I have acknowledged all quotations from the published or unpublished work of other people. I declare that I have also read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences and in accordance with it I submit this project report as my own work.

Signature:

Date:

Academic Year 2012/2013
Abstract

Defeating Memory Error Exploits by Program Diversification and Process Replication

by

Marc Nimmerrichter

Master of Science
in
Information Security

Royal Holloway, University of London
2012/2013

Memory errors have had a major impact on computer security, from the early age of personal computers to the present day. The cat-and-mouse game that has played out between attackers and defenders has provoked substantial research into new attacks and techniques of protection. Although stack-based buffer overflows are often thought of when discussing memory errors, there are also other dangerous memory errors, such as heap-based buffer overflows, format-bugs, integer overflows, and NULL or dangling pointer dereferences. Although major PC operating systems and compilers employ protections like ASLR and canaries, the exploitation of memory errors is still observed regularly in the wild. A promising and more elaborate defence technique is to employ the replication, diversification, and monitoring of processes. There are multiple techniques of diversification, and this work focuses on the diversification of memory. Address-space partitioning would seem to be a very promising approach that allows the deterministic prevention of memory-error exploits that attempt to perform full pointer overwrites. Memory diversification with address-space partitioning causes the original process and its replicas to diverge in behaviour upon exploitation of a memory error. This divergence can be detected by the monitor. A prototype of a monitor program is presented, which utilizes Linux’s debugging functionality with the aim of tracing the diversified processes in user-mode. Issues such as threads, signals, and shared memory are discussed, and potential solutions are presented and analysed. The prototype implementation is generic, and so can be used with any diversification approach. A proof-of-concept demonstrates that such an approach can deterministically prevent the exploitation of memory errors.
Acknowledgement

My project supervisor, Lorenzo Cavallaro, introduced me to the interesting field of memory errors, and deserves a special thank you for our informative conversations and for his support during this work. I am also grateful to Automic Software GmbH for providing me with a professional working environment during this research. Finally, I especially thank my family. Without their support, my studies and this research would not have been possible.
Contents

1 Introduction .......................................................... 1

2 Literature Study .................................................... 3
  2.1 An Overview of the History of Memory Errors ......................... 3
  2.2 Stack Attacks ........................................................................ 4
    2.2.1 Nonexecutable Stack ......................................................... 4
    2.2.2 Canaries ............................................................................. 5
  2.3 Heap Attacks ......................................................................... 7
  2.4 Format String Attacks .......................................................... 7
  2.5 Address-Space Layout Randomization ........................................ 8
  2.6 Null Pointer Dereferences ..................................................... 11
  2.7 Integer Vulnerabilities ......................................................... 11
  2.8 Noncontrol Data Attacks ....................................................... 12
  2.9 The Future ............................................................................ 13

3 Related Work .................................................................... 15
  3.1.1 N-Variant Systems ............................................................ 15
  3.1.2 Diversified Process Replica ............................................... 17
  3.1.3 Multivariant Execution Environment .................................... 19

4 Analysis .......................................................................... 21
  4.1 User- or Kernel mode ......................................................... 21
  4.2 Diversification ...................................................................... 22
    4.2.1 Reverse Stack .................................................................... 22
    4.2.2 Address Space Partitioning ............................................... 24
    4.2.3 Address Space Layout Randomization .................................. 26
    4.2.4 Comparison of Diversification Approaches ......................... 27
  4.3 Monitoring and Synchronization ............................................ 29
    4.3.1 Output Comparison by the Monitor ..................................... 29
    4.3.2 Monitor–Variant Communication ........................................ 30
    4.3.3 Protection of the Monitor ................................................... 30

7
List of Figures

3.1 Default Memory Layout on Linux x86 without ASLR ...........................................16
3.2 Address-Space Partitioning ..................................................................................16
4.1 PUSH on an Upwards-Growing Stack .................................................................23
4.2 RET 8 on an Upwards-Growing Stack .................................................................24
4.3 Exploitability of Reverse-Stack-Execution ..........................................................25
4.4 Issue with Threads Sharing their Address-Space ..................................................36
List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Diversification Approaches</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Firefox—System calls</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>GNU tar—System calls</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Httpd—System calls</td>
<td>35</td>
</tr>
</tbody>
</table>
You need the willingness to fail all the time. You have to generate many ideas and then you have to work very hard only to discover that they don’t work. And you keep doing that over and over until you find one that does work.

An elaborate game of cat-and-mouse has played out over the last three decades between exploiters of memory errors and defenders. Although the mitigation techniques currently implemented in operating systems make it much more difficult to exploit vulnerabilities, attacks are still feasible, and the effectiveness of countermeasures is limited. One problem that was identified several years ago is the lack of diversity in deployed software. A memory-error exploit that works on one machine often works quite reliably on another machine of the same architecture. Canaries and ASLR are approaches to addressing this problem, which work by introducing heterogeneity that makes instances of a program differ from each other. These two approaches certainly harden software against exploitation, but they are not sufficiently elaborate and thus do not provide complete protection. The direction of research aimed at defeating memory-error exploits has not changed much, and the introduction of diversity into programs is believed to be “the solution” to defeating the exploitation of memory-error vulnerabilities. It is important that programmers focus more on preventing vulnerabilities in the first place—for example, by applying program analysis techniques and by using safe programming languages. Yet this is not always possible, and the vast amount of legacy code that has been written over the last decades does not permit abrupt changes. Some time will be needed before such countermeasures are adopted widely. In the meantime, the proposed research approaches include implementing additional and more finely grained diversity to prevent the exploitation of vulnerabilities. The randomization of system call mappings [50], of the instruction-set [52], and of function and variable locations [7] are all means of diversifying instances of programs to probabilistically prevent exploits. These countermeasures can prevent certain attacks, but will always be limited by the entropy provided by the architecture or the operating system and by the need to keep the seed secret. Thus, full protection cannot be guaranteed (as brute-forcing or leaking the seed might be feasible). To increase the likelihood of detecting and mitigating an attack, researchers have proposed replicating processes, diversifying them, and running them concurrently (or in parallel on multiprocessor systems) [14] [10] [21] [75]. Also proposed is diversification of virtual memory in a way that does not rely on keeping secrets. A monitor should synchronize the processes’ interactions with the environment, and thus the processes should behave the same under normal circumstances. Any deviation in behaviour would then indicate an attack, and could be detected by the monitor before any harm could
be done to the system. Such an approach would seem to have the potential of high
detection rates in exchange for some overhead in computation. Since such techniques
appear to be promising (especially for high-security environments) but have not yet been
well-studied, the motivation of this project is to identify whether the approach delivers
comprehensive protection and how it can be implemented. Some research has already
been carried out on this topic in the past, and so this project builds on, analyses, and
improves on those ideas.
Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

Brian W. Kernighan (1942 - )

Prediction is difficult, especially about the future.

Niels Henrik David Bohr (1885 - 1962)

2

Literature Study

2.1 An Overview of the History of Memory Errors

The history of memory errors begins in 1972 [95], when the issue was first publicly discussed in the Computer Security Technology Planning Study [3]. This report addressed security problems faced by the US Air Force that emerged from the sharing of computing and network resources, which had not been practice some years before [3]. The infamous Morris worm that spread in November 1988 was the first large-scale exploitation of memory errors [95] [78] [87]. Among other methods, the worm spread by exploiting a stack-based buffer overflow vulnerability [87]. In response to this malware, the CERT Coordination Center (CERT/ CC) was formed [16]—the first computer security incident response team (CSIRT) [16]. In 1993, the Bugtraq mailing list was set up in response to the lack of up-to-date security-related information available to administrators [81]. In 1995 and 1996, public interest increased with the publication of papers and step-by-step instructions describing the exploitation of stack-based buffer overflows [57] [98] [2], the classic memory vulnerabilities of the time [95].

The first countermeasure to address code injection on the stack was a nonexecutable (NX) stack [95], first implemented by Alexander Peslyak (aka Solar Designer) in 1997 as a patch for Linux kernel 2.0.30. It is often referred to as StackPatch. It only prevented the execution of code on the stack, and malicious code could still be placed in other memory areas and executed by overwriting a return pointer on the stack [31]. A solution to the problem of stack-based buffer overflows was proposed by C. Cowan et al. in 1998 [20]. They described an x86 gcc compiler modification that would place a random “canary” word before the return address of the function prologue. This is checked for correctness in the function epilogue before the function returns. It is assumed that the return address is unaltered if, and only if, the canary is unaltered [20].

In 1999, heap-based overflows were first publicly discussed by Matt Canover and the w00w00 security team [95]. Their sample exploits showed program-specific vulnerabilities that could be exploited using heap-based buffer overflows. They proposed the implementation of a nonexecutable heap as a way to avoid the execution of code injected on the heap. They also suggested compiler-based bound checkers to help developers write secure code [13]. Later in the same year, Tymm Twillman was the first to publicly demonstrate an attack exploiting format-string bugs, publishing an exploit
against proFTPd on the Bugtraq mailing list [91].

The technique of adding randomness to programs, first attempted with canaries, drastically improved with the introduction of Address-Space Layout Randomization (ASLR) by the PaX team in 2001 [95]. The first version only provided support for \texttt{mmap} base randomization, which causes dynamically linked shared libraries to be loaded at different offsets in the address space. Randomization of the stack and Position-Independent Executables (PIEs) followed about a month later [95].

In 2001, the first NULL pointer dereference vulnerability was disclosed [95]. At the time, it was generally believed that such an attack could not cause more harm than a DoS [95]. Some years later, however, it was shown that NULL pointer dereferences may lead to arbitrary code execution [85] [35].

2.2 Stack Attacks

2.2.1 Nonexecutable Stack

Stack-based buffer overflows are usually exploited by inserting shellcode into the buffer and overwriting the return address of the function to point to the beginning of this shellcode. This is possible because the stack grows downwards on the most widely used PC processor architecture, x86, and on others. Thus, a local buffer is placed before the function’s return address. A possible solution to this attack would be to make the stack nonexecutable. As already mentioned, the first nonexecutable stack patch was implemented by Solar Designer for Linux in 1997 [95] [31]. A few months later, he posted a return-into-libc exploit that bypassed his patch on the Bugtraq mailing list. In this attack, he modifies the stack to return execution to a dynamically linked and loaded library function called \texttt{system()}, which then executes program specified as parameter (e.g., \texttt{"/bin/sh"}). This attack was the first return-into-libc exploit, and it shows that even nonmalicious code can be used to produce malicious behaviour. Solar Designer fixed his \texttt{StackPatch} to \texttt{mmap()} shared libraries onto an address containing a zero byte [30]. This prevents the injection of a shared library address in a string-buffer overflow, since a zero-byte indicates the end of a string, and thus causes string operations to halt. Only a few months later, in January 1998, Rafal Wojtczuk (aka Nergal) posted an exploit that bypasses Solar Designer’s \texttt{StackPatch} using the procedure linkage table (PLT) [64]. This table is used for dynamic linking and resides in the .text segment of ELF (executable and linkable format) executables. It points to the shared libraries, so functions such as \texttt{system()} may be called by using the pointers in this table (which do not contain zero-bytes in their addresses), instead of calling the library functions directly. Return-into-lib attacks were later used to invoke the \texttt{mprotect} system call (UNIX) or the \texttt{VirtualProtect} API (Windows), enabling execution in the regions of the stack in which the shellcode resides [95].

In 2000, the PaX team released its own solution to the problem. Besides the nonexecutable stack and the randomization of the \texttt{mmap} base, it also implemented additional protections against return-into-lib attacks. The PaX solution forces data pages to be either writable or executable, but does not allow them to be both. If the processor supports NX, PaX will use it; if not, it will emulate it. In this solution, they introduced the first
form of ASLR [95]. In 2003, the developers of OpenBSD 3.3 implemented various protections against buffer overflow attacks. Among these was what they called W*X, which means that memory that is writable may not be executable, and vice versa. The developers also implemented PROT_EXEC purity, which made execute permission a separate permission, rather than one implied by PROT_READ, although this was not possible for every platform [26] [95]. Additionally read-only data (.rodata) was moved out of the .text section of ELF executables, allowing read-only data to be no longer executable (PROT_READ permissions, only instead of PROT_READ|PROT_EXEC) [24] [95]. This prevents attackers from executing instruction sequences found in static data [26]. Another protection implemented in their compiler was ProPolice, which is a stack-based buffer overflow protection based on canaries.

Following OpenBSD, other well-known operating systems also came to accept the severity of the software vulnerabilities that had been exploited during the preceding years, and set about implementing countermeasures. In 2004, Red Hat improved its security significantly in update 3 for Red Hat Enterprise Linux v3. Its ExecShield project implemented a nonexecutable stack and made use of the NX technology that some processors at the time were already offering to make data nonexecutable. Intel followed soon to implement this feature. ExecShield also implemented ASLR, which randomizes the base of the stack, the heap, and the shared libraries. It also implemented the randomization of the executable itself, where this is supported by the application (i.e. where the executable was compiled as a Position-Independent Executable) [92].

In the same year, Microsoft shipped its SP2 for Windows XP, which contained several security enhancements. One of these was Data Execution Prevention (DEP), which makes use of the processors’ NX feature where available, and emulates it in software otherwise. This means that, as with the solutions of PaX and ExecShield, data can be marked as nonexecutable. The software DEP also hardens Windows’ exception handling mechanism, which has been subject to attacks [62].

In 2005 Sebastian Krahmer published a novel technique, which he called the borrowed code chunks technique [54]. Rather than using entire library functions to jump into, Krahmer uses only short sequences of instructions, called gadgets, from an executable memory area to perform specific operations. Such sequences must end with a return instruction, in order to give control back to the attacker, who controls the stack [73]. With this technique, arbitrary computation can be performed, as long as the required code chunks are somewhere in executable memory [54] [73]. On the x86-64 architecture, arguments are passed via registers and not on the stack, which makes normal return-into-lib attacks harder. This new technique, however, makes manipulation of the registers possible, and introduced a new class of attacks, now called return-oriented programming (ROP) [95]. No bulletproof countermeasures against return-oriented programming have yet been implemented, but bound checkers, to prevent buffer overflows in the first place, and taint-tracking, to distinguish user-input from code, may be possible solutions [95]. Full ASLR can prevent ROP attacks probabilistically.

2.2.2 Canaries

StackGuard was the first implementation of canaries, and was published in 1997 [18]. It consists of a compiler extension that places a random byte value (the canary) just
before a function’s return address. Before the function returns, the value of the canary is checked to see if it has changed—which would indicate a buffer overflow and leads to program termination [18]. StackGuard can only prevent the overwriting of a return address by direct buffer overflow attacks, where memory corruption happens linearly and sequentially [95]. Unfortunately, vulnerabilities that allow arbitrary memory corruption (such as format-string vulnerabilities that provide an attacker with a write-anything-anywhere primitive) mean that the return address can be altered without changing the canary value. This issue was addressed by StackShield [88], another compiler extension released in 1999 [95]. This tool does not work with canaries, but rather focuses on the return address itself. In the function prologue, the return address is copied from the stack to an array in the .data memory area. Then, in the function epilogue, the return address on the stack is replaced with the address stored in the .data memory area. This means that an overflow cannot redirect execution by manipulating the return address.

Over the following years, different attacks on StackGuard and StackShield were demonstrated [11] [72], leading to further evolution in these protections. To mitigate a nonlinear return address overwrite (with a write-anything-anywhere primitive), StackGuard was improved to form the canary by XORing the random value with the return address. [11] shows that StackGuard cannot prevent GOT overwrites, which can be used to redirect the execution flow in a way similar to overwriting the return address. As can be seen in [72], protecting the return address and leaving the saved frame pointer (SFP) unprotected is not sufficient, and gives attackers additional means to bypass these countermeasures. In 2000, a paper presented Stack Smashing Protection (SSP), also called ProPolice. This idea is based on StackGuard, but avoids its shortcomings by re-arranging the variables on the stack. Since it places all arrays directly before the canary value, linear overflows cannot do any damage to local variables, to the SFP, or to the return address [39]. This technique was implemented as a patch for gcc version 3, and was included in the mainstream code of version 4.1 [95].

Microsoft also introduced a protection similar to StackGuard with its compiler suite of 2003 [55] [61]. The protection could be enabled by setting the /GZ flag, which later became a default in the suite [59] [42]. In 2003, David Litchfield presented a novel technique for bypassing Microsoft’s stack protection by manipulating pointers of the _EXCEPTION_REGISTRATION_RECORD structure. This structure is located on the stack, and implements a linked list consisting of a pointer to the next record and a pointer to the exception handler. A stack-based buffer overflow can be used to overwrite the pointer to the handler, and thus to redirect the flow of execution [55]. Such techniques are called structured exception handling (SEH) exploits [95]. Following this publication, countermeasures were proposed and implemented. One solution was to implement the /SAFESEH linker flag, which makes a program immune to such attacks, but requires rebuilding the application. When this flag is set, the exception dispatcher checks whether the handler appears in the image-specific list of safe exception handlers [86] [60]. For some architectures, all exception handlers are already noted here, and so this flag does not need to be (and cannot be) used [60]. Another approach is structured exception handling overwrite protection (SEHOP), which terminates the linked list with a symbolic termination record. When an exception is raised, the integrity of the linked list can be checked by following it to the termination record. This countermeasure does not require the application to be rebuilt, but only works well when ASLR is enabled [63] [86] [95].
2.3 Heap Attacks

In January 1998, the first heap-based buffer overflow was published [33], and this was followed one year later by a paper that described this in more detail [83]. These publications, which discussed application-specific exploitation techniques, showed that buffer overflows are not limited to the stack. In 2000, Solar Designer reported the first description of a more advanced, generic heap-based memory error exploit [32], which was possible because the in-band metadata of the memory manager (malloc) were not checked for integrity [95]. [4] gives a detailed description of Solar Designer’s technique: It explains that memory managers usually chop the heap up into so-called chunks, which are allocated by the program. The main issue is that metadata such as the chunk sizes, the “in-use” flag, and the pointers to the next and previous chunks are stored in-band (with the data). This means that a buffer overflow in such a heap-chunk can not only override the data that follow, but also the control information kept by the memory manager about the chunks. As described in detail in [4], the free() function of GNU C’s malloc performs the following operation to coalesce the next chunk (if free) with the one that is to be freed:

\[(\text{next} \rightarrow \text{fd} + 12) = \text{next} \rightarrow \text{bk}\]

where next is the chunk following the one containing the vulnerable buffer, and fd and bk are respectively the pointers to the next (forward) and to the previous (backward) chunks. Since the fd and bk of next can be manipulated, this statement provides an attacker with a write-anything-anywhere primitive. An attacker could craft a buffer to overwrite next->bk with the address of malicious code (possibly at the beginning of the buffer if the heap is executable) and next->fd with the location of a pointer, such as the return address, a GOT entry, a function pointer, or any other that could be used to redirect the execution flow.

Over the following years, further documentation and more advanced techniques appeared discussing generic heap-based buffer overflow exploits for Unix and Windows systems [40] [56] [51]. With the release of SP2 for Windows XP in 2004, Microsoft introduced a nonexecutable heap, heap cookies (a canary-like protection technique), and safe heap management data unlinking. As Canover [12] explains, the canary length of 8 bits is insufficient to defeat brute-forcing. Safe unlinking, which performs the following integrity check:

\[B \rightarrow \text{Flink} \rightarrow \text{Blink} = B \land B \rightarrow \text{Blink} \rightarrow \text{Flink} = B\]

can also be evaded under certain conditions [12]. Other techniques and mitigations appeared in the next few years [95]. Unfortunately, nonexecutable heaps can be defeated by return-into-lib-like attacks. Heap spraying, which is explained later in this paper, can also be used to bypass ASLR [95].

2.4 Format String Attacks

Many programming languages, including C, support so-called variadic functions (often also referred to as variable argument or vararg functions). Such functions can take a dynamic number of arguments passed on the stack, and it’s left to the function itself to
interpret the content of the stack [19]. The printf functions of C’s stdio library are an example of this: they usually take a format string as the first parameter, containing static text and format directives (such as placeholders for variables). Following this first parameter are the variables that need to be provided (e.g., to fill the placeholders). Since the number of directives in the format string can differ from function call to function call, the number of arguments can also differ. Programmers make mistakes, and so user-provided input is often used as format-string parameters.

// Insecure:
printf(input);

// Secure:
printf(“%s”, input);

This means that an attacker could make use of a format bug by placing format directives in the input string, leading to results not intended by the programmer [19]. Placing %x directives in the format string, for instance, would pop items off the stack. The %n directive writes the bytes written so far into the memory address provided as an argument on the stack. A format bug could lead to the leakage of information, such as cryptographic keys, from a vulnerable program’s address space [95], but by making use of printf’s %n directive, it could also provide an attacker with a write-anything-anywhere primitive [19]. This is especially dangerous, since it allows very precise overflows to be executed on arbitrary memory locations, and can thus be used to bypass canary protection (except where StackShield or XOR-canaries are used). In 1999, the first format-string vulnerability was disclosed for proftpd [91]. In the following years, general descriptions of this type of vulnerability were published [66] [80].

In 2001, FormatGuard [19] was proposed. This is a static analysis approach to preventing format bugs that uses C preprocessor scripts and wrapper functions. These latter count the number of arguments expected and compare it to the number of arguments defined in the format string; if they differ, the process terminates with a syslog entry. FormatGuard is a static analysis approach, and thus leaves some vulnerabilities [95]. In 2012, Microsoft disabled the %n feature by default in its Windows C Runtime (CRT) [95]. The GNU C library also implemented some protection with its FORTIFY_SOURCE patches, which can be configured to allow %n in format strings only when the string is stored in read-only memory [49]. In general, format bugs can be detected and fixed easily, and so “the low handing fruit has been harvested long ago” [95].

2.5 Address-Space Layout Randomization

To exploit a memory-error vulnerability, knowledge of the target-process’ address space is required. Randomizing this address space is thus a probabilistic approach to hardening programs against exploitation. An attempt to exploit a program with “perfect” Address-Space Randomization would result in a crash with a quantifiable probability [95]. In 2001, the PaX Team were the first to propose this type of protection [95]. Their initial patch for the Linux kernel randomized the mmap base (affecting dynamically linked libraries: only the first library’s location is randomized, and others are mapped after it), which made return-into-lib attacks more difficult. Only a few months afterwards,
randomization of the stack base and support for Position-Independent Executables (PIEs) followed [95]. Stack base randomization makes it harder for an attacker to find injected code or data [95]. PIEs are, like dynamically linked libraries, position-independent, and can thus be placed at a random position in a process’ address space. This is important, as the randomization of dynamically linked shared libraries can be bypassed by using an ELF’s PLT and GOT [65], which resolve the addresses of these dynamic libraries. Such attacks are called return-into-plt attacks, and can be protected against by using PIEs, because if the executable is randomized, the PLT (which is part of an ELF executable) is also randomized. The importance of PIEs is again shown in research from 2009 that used return-oriented programming to retrieve or modify addresses in the GOT [74]. In 2002, PaX proposed randomizing the kernel stack, since attacks on this had become more popular [95]. In the same year, an attack on PaX’s full ASLR was published: in this, a buffer overflow causes a format-string bug to leak absolute memory addresses, which can then be used to perform a return-into-lib(c)-like attack [36]. To make the randomization complete, PaX implemented the randomization of the brk (heap) base in the following year.

The first operating system to implement such concepts in its mainstream kernel was OpenBSD, in its 3.3 and 3.4 releases of 2003 [24] [25]. Apart from the implementation of WE, ProPolicy, and -rodata, these versions also implemented stack gap randomization (which places a random gap in the stack to randomize stack addresses) and the randomization of dynamically linked shared libraries (mmap) [26]. However, to avoid conflicts between OpenBSD and the POSIX standard, kernel stack randomization was not implemented [95]. In 2004, with version 3.6 of the operating system, OpenBSD also implemented randomization of the heap by making malloc() use mmap() instead of sbrk() [27]. Version 4.4, released about 4 years later, also supported PIEs [28], and the following release was shipped with the necessary gcc features for compiling PIEs [29].

With Enterprise Linux version 3, update 3, in 2004, Red Hat improved its ExecShield by implementing ASLR. It randomized the stack, heap, and shared libraries. It also added support for PIEs and contributed to the GNU Compiler Collection so as to make compilation of an executable as a PIE possible [92].

With version 2.6.12-rc1, the Linux kernel also began implementing ASLR. First, only stack and mmap base addresses were randomized [94] [93], but with kernel version 2.6.25, released 3 years later, support for PIEs and heap base randomization was added, making ASLR complete [95]. Linux does not support kernel stack randomization [95]. An attack specific to older Linux versions (up to 2.6.17/20) made use of a nonrandomized virtual dynamically linked shared object (VDSO)—the Linux gate [22]. This again shows the importance of full ASLR, in which every part of a process’ address space is randomized.

In Windows Vista Beta 2, released in 2006, Microsoft introduced ASLR into its OS, supporting stack, heap, and library randomization [45] [95]. Following the release, researchers published papers describing weaknesses in this implementation of ASLR [70] [96], some of which were refuted by Microsoft [46], and others acknowledged [95]. The main weaknesses of the ASLR in Windows Vista are caused by performance optimizations, which result in lower entropy than in the case of other implementations [70] [96]. Research from 2012 [97] and 2013 [53] has also shown Windows 7 and 8 to be vulnerable under low memory conditions. Research conducted in 2010 [68] showed that third party applications are adding support for ASLR only slowly by compiling their
code as a PIE.

Since 2007, Apple has also randomized library addresses in its MAC OS X 10.5 release [47]. It took until 2011 for full ASLR to be supported in the 10.7 release [41].

ASLR as implemented in current operating systems is coarse-grained, as only the base addresses are randomized, while the relative offsets of objects within the memory region are static. Thus, if an attacker can gain the virtual address of any object in the memory area of interest (such as a library or the stack), then the addresses of all other objects in the memory area are revealed. Apart from that, such coarse-grained randomization could be exploited by partial address overwrites, in which only the offset is overwritten, but the randomized part of the address stays untouched [36]. To overcome these vulnerabilities, researchers have proposed fine-grained address-space randomization by binary-rewriting [6] or source-to-source transformation [7], but such protections have not yet been implemented widely.

In 2004, research showed that randomization on 32-bit systems suffers from architectural constraints, which make brute-forcing memory addresses possible [82]. Another approach for bypassing ASLR is to use heap spraying. Such attacks, which are usually conducted using scripting languages like JavaScript, allocate many large objects on a process’ heap. These objects contain x86 instructions that form a NOP-sled followed by shellcode. This increases the probability of hitting one of the NOP-sleds during pointer manipulation [71]. DEP and similar NX heap technologies make such attacks harder, but not all programs use these, and attackers have even managed to turn off such protection before executing their malicious code [71] [8]. Some methods for detecting malicious content in script objects were proposed in 2009 [71] [37]. In 2010, Blazakis [8] showed how pointer addresses can be inferred in dynamically typed scripting languages, even if ASLR is enabled by exploiting the implementation details of the interpreters. In the same paper, JIT spraying was introduced, which exploits the dynamic code generation of just-in-time compilers. Such compilers need to generate code at runtime, and thus store the code with execute-permissions on the heap. As we have learned from return-oriented programming, a harmless sequence of instructions can become harmful if the instruction pointer is placed in the middle of an instruction, instead of at its beginning. This means that benign-looking scripts can become harmful if they are compiled at runtime and execution is forced to start at another offset. JIT spraying is a technique that loads such bytecode into a program’s heap multiple times. Pointer inference and JIT spraying together provide an attacker with the means to bypass DEP and ASLR [8].

The idea of ASLR is to introduce diversity into programs as a probabilistic approach to hardening systems. Unfortunately, the method relies on keeping secrets, which makes information leakage a problem [36]. Its probabilistic nature also makes it vulnerable to brute-force attacks, especially on 32-bit systems [82]. The idea of ASLR is further constrained by its implementations, which only change the base addresses, and leave static offsets as a potential vulnerability [95]. Nonrandomized memory regions (as in the case of non-PIE executables) will always be an easy target for attackers [65] [74].
2.6 Null Pointer Dereferences

NULL pointers are used in low-level programming languages like C to represent specific states, such as the ends of linked lists. To avoid the problems of NULL pointer dereferences, OSes usually do not map the first page [95], which would otherwise make such references result in errors. This makes NULL pointer dereferences difficult to exploit [95]. However, research has shown that specific conditions can lead to a NULL pointer dereference vulnerability that is exploitable.

In 2006, research [85] was published that demonstrated the exploitation of a NULL pointer dereference in Internet Explorer. A NULL pointer dereference on Windows will result in UnhandledExceptionFilter (UEF) being executed. UEFs are expected to be set and

\[
\text{DLL1 sets UEF and stores previous UEF}
\]
\[
\text{DLL2 sets UEF and stores previous UEF}
\]
\[
\text{DLL2 restores previous UEF}
\]
\[
\text{DLL1 restores previous UEF}
\]

restored symmetrically, as follows:

However, forcing DLLs to be unloaded asymmetrically can lead to an asymmetric restore of UnhandledExceptionFilters, resulting in a pointer referencing the memory area previously occupied by the now unloaded first DLL. This can be exploited by placing malicious code in this memory area through, for example, heap spraying [85]. On ARM systems, the memory address 0x0 is mapped, and contains the exception vector table. If a user can supply data that is written at a memory address referenced by the vulnerable pointer, a NULL pointer dereference will allow the exception vector table to be overwritten [48].

In 2008, an exploitation technique for Adobe Flash (x86) was shown. By crafting a special SWF file with shellcode, ActionScript code, and an invalid scene_count value, a NULL pointer dereference can be triggered and exploited. The attack makes use of a vulnerability in the bytecode verification of the ActionScript Virtual Machine [35].

In the following year, attacks on the Linux kernel were published [89] [90] that exploited the possibility of mapping the 0th page. This can occur if the user has the CAP_SYS_RAWIO capability. setuid programs owned by root have this capability. It turned out that SELinux “hardened” versions were similarly (and under certain circumstances, even more) vulnerable to such attacks [38].

2.7 Integer Vulnerabilities

Although integer vulnerabilities are not memory errors, they can cause such errors. They occur because of the way numbers are represented in memory. [9] and [23] classify integer errors into the following types: truncation problems between differently sized integers, wrap-arounds when numbers are too high or too low, and signedness problems that result from the way negative numbers are represented in memory. If an integer vulnerability results in a memory error, a memory overwrite, an information leakage [44], a denial of service, or a logic error [23] can all occur. The majority of integer errors can lead to arbitrary code execution [23].

Consider a function that takes an unsigned int i (32 bit on x86) as argument, and assigns it to a short unsigned int s (16 bit on x86). This will have the effect of
the most significant 16 bits of \( i \) being ignored, and the 16 least significant bits being stored in \( s \). If size checks are performed on \( s \), but \( i \) is taken as the length parameter in memory operations (string copy, etc.), the check could pass despite the fact that the value is invalid (for example, if the value of \( i \) is too large, it is not represented in the short unsigned int, since the most significant bits have been lost during conversion). This is a typical example of a conversion problem, and can lead to a buffer overflow.

Wrap-arounds are similar, but occur even without type conversions. Consider a function that checks whether the size of an input buffer (provided to the function as an unsigned int) is smaller than the target buffer. Since the function wants to append a NULL byte after copying the buffer, it first checks if \( i+1 < \text{\textbf{buffer}} \text{\textsize{size}} \), and then copies \( i \) bytes from the input buffer into the target buffer. If the maximum integer value (232) is provided as the size of the input buffer, adding 1 will cause a wrap-around (since the resulting value would have 33 bits, which cannot be stored in a 32-bit integer, only the 32 least significant bits are taken into account, giving the result modulus 232 + 1). This will be 0, and will thus pass the size check. Following this addition, \( i \) bytes are copied into the target buffer, causing a buffer overflow.

Signedness problems occur when operations that interpret an integer as signed are followed by others that interpret it as unsigned, or vice versa. Consider a function that copies the number of bytes provided by a signed int \( i \) from an input buffer to a target buffer. First, it is checked if \( i < \text{\textbf{size\text{\textsize{of}}} (\text{\textbf{buffer}})} \), and if so, memcpy is called with the buffers and \( i \) as parameters. The problem here is that the comparison interprets the integers as signed, whereas the memcpy treats them as unsigned. So if a negative number is provided for \( i \) (e.g., 0xffffffff = -1), then the if statement will be true, but memcpy will result in a buffer overflow (because it will treat 0xffffffff as the very high positive value 232 - 1). Signedness errors are harder to exploit, since such high values are likely to use too much memory and thus result in a segmentation fault [9].

Integer overflows were first publicly discussed at Black Hat 2002 [34], and were well-documented later in that year [9]. The same issue of Phrack also published a protection technique against signedness vulnerabilities that assumes that loops usually do not iterate as often as required for signedness integer problems to be dangerous. It thus suggests compiler modifications to add checks for such high numbers of iterations [44]. Several, more elaborate protections were studied during the following years [95].

2.8 Noncontrol Data Attacks

Control data can be defined as data that are loaded into a processor’s program counter (the eip register on x86) at some point during execution [17]. Since attacks on control data are often feasible, and since exploitation recipes are generic for a range of applications on a particular architecture, attacks on noncontrol data are not often seen in the wild; they usually take more effort to perform and are application-specific [17]. Yet this also means that our defence measures against the exploitation of memory errors are tailored to control data attacks [17]. There are few widely implemented protections against noncontrol data attacks. The best countermeasure would be to avoid memory error bugs in the first place, but this stage has not yet been reached (see Chapter 2.9). Chun et al. [17] identify the following types of security critical noncontrol data:
Configuration data
User input
User identity data
Decision-making data
File descriptors
RPC routine number

One attack that Chun et al. [17] managed to perform against WU-FTPD exploits a format-string vulnerability to manipulate a data structure that stores the user's UID. In their attack, the UID field in this data structure is set to 0. When the FTP client accesses a file subsequent to this attack, WU-FTPD attempts to impersonate the user by issuing the seteuid system call. The FTP server consults the data structure to retrieve the user's previously manipulated UID, which is then passed to the seteuid system call as an argument. This means that the server now runs with the effective UID 0, instead of with the user's UID, allowing the attacker to manipulate arbitrary files, including `/etc/passwd`, to create an additional user account as backdoor.

Another attack performed by the same researchers manipulates the CGI path of an HTTP server to execute a shell in place of the intended executable for data processing (such as `php`) [17].

As already mentioned, protection against memory error exploitation is often focused on control data attacks. Whether a specific noncontrol data attack is defeated by a certain protection technique depends very much on the attack (on which vulnerability is exploited and where the data is located). One countermeasure that can be implemented by developers without the need for any additional tool is to reduce the lifetime of security-critical data. If the data has not been initialized when the memory error bug occurs, or if it has already been deallocated, then the bug cannot be exploited successfully. This might not be easy to ensure in all circumstances [17].

2.9 The Future

The peak of memory vulnerabilities disclosed relative to the total number of disclosures occurred in 2004, with about 25%, but the value decreased to about 15% in 2010 [95]. There could be several reasons for this, but one is probably that vulnerability research is shifting towards the web [95]. Yet the amount of code that is still written in unsafe languages implies that we will not rid ourselves of memory errors in the near future. The past has shown that we should always assume that a piece of software will be exploited once, and our environment should be prepared to minimize the consequences of an intrusion. This will still hold true in future. Network administrators have a difficult job in this regard, since it is nearly impossible to keep attackers out, as our daily newspaper of choice reminds us. As Bruce Schneier puts it, "The odds favor the attacker: defenders have to protect against every possible vulnerability, but an attacker only has to find one security flaw to compromise the whole system." [79]. More robust detection and prevention systems that help organizations and individuals to protect their assets are still required.
If you want to build a ship, don’t drum up the men to gather wood, divide the work, and give orders. Instead, teach them to yearn for the vast and endless sea.

Antoine De Saint-Exupery (1900-1944)

Related Work

3.1.1 N-Variant Systems

In 2006, Cox et al. [21] described what they called an “N-Variant System”. This framework, which the researchers implemented for Linux, replicates a process and diversifies the resulting processes using address-space partitioning and instruction-set tagging. The goal is to achieve secretless deterministic protection.

Address-space partitioning is performed by modifying the linker scripts used by GNU ld during compilation of the variant, and is enforced at runtime by limiting the size of both processes’ data segment using ulimit. Partitioning ensures that the variants use disjoint parts of the virtual address space, which guarantees that a memory address valid for one process will not be valid for the other one. Imagine that process P0 uses memory addresses beginning with 0, whereas the memory addresses of process P1 have 1 as their high-order bit. This means that a pointer overridden by an attacker with a buffer overflow, for instance, will only be valid in one of the variants (processes). In the other variant, it will result in a segmentation fault, and thus in termination by the operating system. Such a termination, which indicates a compromise, would be detected by the monitor code that was added to the do_exit() kernel function (executed by all terminating processes), thus allowing the other variant to be terminated, too. This kind of address partitioning allows the detection of all full pointer overwrites deterministically (in contrast to ASLR with its probabilistic property).

The N-Variant model consists of a polygrapher that synchronizes the input from the environment to the processes and a monitor that checks that the outputs of both processes are equivalent. Both are implemented as one process. If the output (the system calls or system call parameters) of the processes differ, then their execution has diverged, which indicates a compromise (or else a false positive due to nondeterminism issues). This means that a compromise might not be detected immediately, but only at the next synchronization point (at system call granularity). Since a process should not be able to change the state of the machine without issuing a system call, a compromised process cannot do any harm. The researchers implemented the system call synchronization by modifying the Linux kernel and writing system call wrapper functions [21].

Instructions are tagged using a binary instrumentation tool. Before each instruction, a tag specific to the process is added. In the original authors’ implementation, the tag was one byte for simplicity of implementation, but in principle the tag can be thought of
Figure 3.1: Default Memory Layout on Linux x86 without ASLR

Figure 3.2: Address-Space Partitioning
as a bit (so, a 1 before each instruction of P1 and a 0 before each instruction of P0). At runtime, a software dynamic translation tool (a kind of virtual machine) checks whether the tags are correct for this particular instance, and if they are, it removes them and prepares the instructions for execution without the tags. If the tags are invalid, an alarm is raised and the execution is terminated. Thus, if an attacker injects malicious code into the N-Variant System's processes, the tags preceding the instructions can only be valid for one variant, but not for the other. This should prevent attackers from injecting malicious code [21].

Some nondeterminism issues, such as signals, threads, and shared memory with write permissions remain unsolved. Asynchronous signal delivery could cause the variants to diverge, hence resulting in false positives. Nondeterministic scheduling of threads could lead to the same result. Mapping shared memory with write permissions would allow a process to change the state of the machine (communicate with the environment) without monitoring [21].

The researchers also provided some interesting performance statistics. The environment used for benchmarking was a single-core 2.2 GHz Intel Pentium 4 machine with 1 GB RAM and Fedora Core 3 (with the 2.6.11 Linux kernel) as operating system and Apache as web server. Their benchmark system was WebBench 5.0. On an unsaturated machine, replicated execution with address-space partitioning caused a 18% increase in latency and a 14% decrease in throughput, whereas on a saturated machine, latency increased by 94% and throughput decreased by 48%. The latter measurement is reasonable, since all computation has to be done twice and processes also need to wait for each other at the synchronization points (system calls). Instruction-set tagging caused an increase in latency of 28% (unsaturated machine) or 173% (saturated machine), and a decrease in throughput of 24% (unsaturated machine) or 63% (saturated machine) [21]. All measures are rounded to the nearest integer.

3.1.2 Diversified Process Replicas

Cavallaro et al. [14] [10] independently and simultaneously researched a similar idea, which should likewise provide secretless deterministic protection against certain kinds of memory error. As well as address-space partitioning (as applied by the N-Variant System approach), they implemented address-space shifting of the replicated process to achieve a probabilistic countermeasure against an additional kind of Impossible Path Execution (IPE), which can be invoked by overwriting a pointer's least significant byte (also referred to as partial address overwrite). They also proposed a solution to the problem of writable shared memory, as well as to synchronous signal delivery. The implementation of this approach in user space is probably the most important difference to the N-Variant System. In their paper, they refer to the replicated process as replica (P_n), in contrast to the original process (P); the monitor is referred to as the Tracer (T).

The address-space partitioning is achieved by a modified ld linker script, which is used to compile the replica to modify the location of the .text, .data, and .bss segments. Furthermore, the runtime dynamic linker ld-\texttt{linux}.so was diversified and also modified to move the stack and m\texttt{map} areas. The result of this is that P and P_n run in disjoint virtual memory locations, which ensures detection of a full address overwrite.
deterministically, as previously described. To ensure that the virtual memory remains partitioned at runtime, the memory reserved for the other process is mapped with no privileges (PROT_NONE) [15]. This is a more elaborate protection than in the case of the ulimit applied by Cox et al., as it ensures that the address space stays partitioned at runtime. ulimit is capable of limiting a process’ data, stack, or virtual memory size, but it does not guarantee the partition to which those segments are mapped. Although ulimit seems to be sufficient under normal circumstances, an attacker might find a way of remapping some of these areas to memory addresses in the other variant’s partition.

As a countermeasure to partial address overwrites (modification of the least significant byte), the ld linker script was further modified to insert k bytes of “junk” data at the beginning of the replica’s address space. This means that a buffer overflow attack that overwrites only the least significant byte of a pointer (easily achieved on little-endian systems), would make a different relative jump on P and P_r. This will either cause an exception (because of an illegal instruction or bus error) or will lead to both executing different instructions. The latter case is likely to make P and P_r diverge in behaviour, which can be then detected by the tracer. It is clear in this case that system call parameters need to be compared as well. In the case of an LSB overwrite, it is likely that the variants diverge in behaviour without a segmentation fault. In the N-Variant System, which does not perform address-space shifting, an LSB overwrite would lead to both variants executing the same impossible path, and thus detection by comparing system calls would not be possible. For shared objects, where the shifting cannot be performed at compile time (say, if source code is not available), the shifting is more difficult to implement. In their implementation, Cavallaro et al. [14] [10] require the source code to be available for shifting at link time, but some ideas on how to implement this at runtime are also outlined [14] [10].

The tracer uses ptrace to monitor the process and its replica. This system call allows the tracer to attach to the process; it is then invoked after entering a system call and before the system call returns (both are referred to as the processes’ rendez-vous points). The monitor’s job is to compare the behaviour of P and P_r, and, if it differs, to terminate both of them; it also synchronizes system calls, which are either executed, simulated, or specially treated.

Writable shared memory is made possible by making use of the CPU’s page fault mechanism. Signals are synchronized by delivering them at the rendez-vous point. Some issues remain, however, and are discussed in later chapters.

The performance penalty with this approach was measured using a thttpd web server running the httperf benchmark on a single-core 1.3 GHz Intel Centrino with 512 MB RAM and GNU/Debian Linux (kernel 2.6) as operating system. Since thttpd optimizes performance by memory-mapping file-system objects (mmap) to avoid read system calls and also by caching, throughput decreased by only 1.2%, and latency increased by 31%. To obtain more general results, the researchers modified thttpd to disable caching and to make it perform read system calls, instead of using mmap. This caused a throughput decrease of about 44% and increased latency by about 600%. This performance penalty is attributed to the inefficiency of ptrace for copying data. Cavallaro et al. mention that an implementation that instead uses other methods could improve performance [14] [10].
3.1.3 Multivariant Execution Environment

Some years later, Salamat et al. [77] [75] published their work on the “Multivariant Execution Environment”. Their idea is similar to that of Cox et al. and Cavallaro et al., as it also tries to prevent the exploitation of memory error vulnerabilities by parallel execution of diversified processes. However, a closer look reveals that there are significant differences in the implementation here. Although they try to describe a very generic user-space system that can run multiple processes diversified in different ways, the researchers describe their implementation, which applies stack-direction reversing as a diversification technique, in more detail. In their papers, solutions to problems of nondeterminism, such as signal delivery and threading, are proposed. Monitoring is done at the system call level, as in the other approaches. The monitor checks and compares the system calls issued by the variants and synchronizes the results.

Reversal of stack direction is performed at compile time and implemented by library and compiler (gcc) modifications [76]. In a Multivariant Execution Environment, at least one process runs with a downwards-growing stack (normal under x86), and at least one other runs with an upwards-growing stack. A buffer overflow would overwrite different data in both variants. In the process, with a downwards-growing stack, the buffer overflow would overwrite data previously pushed on the stack, since memory is nevertheless written from lower to higher addresses. Such data could include local variables, the saved frame pointer, the return address, and parameters. In the upwards-growing stack, the overflow can only overwrite local variables or unallocated memory. The different consequences of a memory error here make both variants differ in their behaviour, and this is detected by the monitor. The technique can prevent attacks that exploit memory errors on the stack and overwrite control or noncontrol data. Unfortunately, it does not provide any protection against memory error exploits on other segments of a process’ address space, such as the heap.

Monitoring happens at system call granularity, and is implemented using ptrace. Most system calls are executed by the monitor, though where necessary, they are executed by the variant processes themselves. The opposite approach was taken by Cavallaro et al. in their implementation.

The researchers evaluated the performance of different approaches to monitor–variant communication. As indicated by Cavallaro et al., they also show the inefficiency of ptrace for copying data, and compared its performance with FIFOs (named pipes) and shared memory. The best performance could be achieved using shared memory for all data larger than 40 bytes. Shared memory is about 900 times faster than ptrace, and about 20 times faster than FIFOs for 128 kilobyte data. Thus, their implementation uses ptrace only for data smaller than 40 bytes, and shared memory otherwise. Shared memory, however, introduces some security concerns, which are evaluated in Chapter 4.3.2.

The researchers assume that it is good enough to deliver signals to both processes between the same two system calls (called signal time frames), instead of at the exact same point of execution. This assumption might hold true for most programs, but could break some that behave differently depending on the specific state of execution the signal is delivered at. Delivery within the same signal time frame is achieved by code injection with signal and system call delaying. A new monitor is spawned for each thread, which
the researchers claim as a solution to the threading problem. Yet this might not hold true for all programs, and is discussed in more detail in Chapter 4.5.

Different performance benchmarks were executed on an 2.5 GHz Intel Core 2 Quad Q9300 running Ubuntu Linux 9.04 with the 2.6.28-11 Linux kernel. 18 applications were tested (gzip, gcc, apache, tar, find, equake, mesa, ...). These had a performance penalty of less than 10% on average when running two monitored variants (one diversified with stack-direction reversing) on an unsaturated CPU. On a loaded system, the average decrease in performance was about 50%. Their tests revealed that I/O, not CPU, is the bottleneck for some programs, which means that they perform nearly identically on saturated and unsaturated systems (these programs were apache, tar, and find) [77] [75]. Unfortunately, due to the different benchmarking techniques and metrics, comparison of the performance with the approaches described earlier is difficult.
4.1 User- or Kernel mode

One of the fundamental questions to be investigated, and one which greatly influences further analysis, is whether such a system should be implemented in user mode or in kernel mode. The x86 and x86-64 architectures support four different privilege levels (also called “rings” or “modes”). Most operating systems today make use of two of them: ring 0 for the kernel and drivers, and ring 3 for all other applications. Code executed in ring 0 has full privileges, which means access to privileged instructions and direct access to shared resources (such as direct access to memory, and thus access to the memory of other applications). Code executed in ring 3 does not have access to some instructions and is limited to its own virtual memory space. Access to shared resources must to be done through system calls. Such system calls invoke code executed in kernel mode, which accesses the resource on behalf of the process, after checking whether the invoking process is authorized.

The choice of user mode or kernel mode for running a replicated and diversified execution system makes significant differences for the limitations, compatibilities, security, performance, and difficulty of deployment. The researchers behind the N-Variant System decided on a kernel-mode implementation, whereas the implementers of Diversified Process Replica(DPR) and the Multivariant Execution Environment (MVEE) opted to implement their system in userland.

An implementation in kernel mode has the advantage that kernel code and data structures can be manipulated directly. For example, wrappers around system calls and a monitored do_exit() function can be implemented easily from kernel mode. A kernelland implementation would also offer more possibilities of runtime diversification, which would be difficult to achieve in userland. Monitoring and synchronizing the variants is likely to be more efficient in kernel mode, as no restrictions are enforced there on accessing other processes’ memory. However, research into DPR and MVEE have shown that a userland implementation is possible, and only a few nondeterminism problems remain unsolved [75] [14] [10].

Since the kernel needs to be recompiled with the modifications, maintenance and deployment of a kernel-mode implementation is more difficult in contrast to a userland one [76]. In the case of the latter, no recompilation or reinstallation of the kernel is
needed, but only installation and configuration of a userland application. Easier deployment will increase acceptance by users.

Another significant disadvantage of an implementation in kernel mode is the decrease in the system’s security [76] arising from the increase in the size of the Trusted Computing Base (TCB). The TCB is “a core set of functionality that is assumed secure” [43] and consists of “hardware, firmware, and a basic set of OS services that allow each application to protect and secure its data and execution” [43]. The idea of trusted computing is to have a chain of components that rely on each other, and thus need to trust each other. A component higher up in the chain trusts the lower ones and expects them to work correctly. Security restrictions are enforced by the TCB, which needs to be bug-free in order to guarantee this. A compromise of the TCB means a compromise of the whole system. Processes should be isolated from each other by the TCB, and access to shared resources should be controlled strictly. Thus, a compromise of code outside the TCB can only act within the boundaries of the process’ privileges, and any damage can be kept under control (theoretically). Since the number of bugs increases with code size, keeping the TCB as small as possible has a positive influence on system security [58].

Since the main incentive behind replicated and diversified execution is to improve security, a userland system is the preferred choice. Future analysis will thus be made on the basis of such an implementation.

4.2 Diversification

Different methods for program diversification have already been proposed. The following paragraphs should provide an overview of those ideas.

4.2.1 Reverse Stack

Salamat et al. [76], who worked on diversification for the MVEE system, employed reversal of the stack direction to diversify their variants [75] [76]. On the x86 architecture, the stack typically grows downwards (towards lower memory addresses). By modifying the compiler and applying some basic pointer arithmetic, the stack direction can be changed to grow upwards (towards higher memory addresses). gcc was modified at the register transfer language (RTL) intermediate representation level. [76].

At first glance, it might seem logical to simply replace stack operations like PUSH and POP with ADD/SUB and MOV; however, a closer look reveals that this would cause problems. ADD and SUB do not only change the operands, but also alter the CPU flags. Inserting such instructions could thus change program semantics [76]. Since PUSH and POP can also be called with an indirect operand, they cannot be replaced by MOV. PUSH and POP do not only utilize their operand, but also internally use ESP, which contains the address of the top of the stack. MOV is not an instruction specifically for the stack, and so it requires two arguments to specify source and destination. Since it does not make use of ESP internally, it must be specified as operand. But replacing PUSH (%eax, %esp) with MOV (%eax), %esp is not possible, as MOV does not allow more than one indirect operand. Facilitating a temporary register would be possible, but this is not efficient, so other instructions were chosen to replace stack operations. The researchers use LEA
Figure 4.1: PUSH on an Upwards-Growing Stack

![Diagram showing PUSH on an Upwards-Growing Stack]

Instead of ADD/SUB, since it does not modify the CPU flags. Instead of MOV, they use the original stack operations—PUSH or POP [76].

In a downwards-growing stack (normal on x86), the stack pointer indicates the beginning of the last element that was pushed. To allocate space for the next element, the stack pointer (esp) is decreased by the element’s size in bytes. In the upwards-growing stack implemented by Salamat et al., the stack pointer points to the next free slot, and this requires adjustments to esp before and after each stack operation [76]. Hence, PUSH (%eax) will be replaced with the instructions illustrated in Figure 4.1.

The stack pointer initially indicates the next free slot on the stack (0). Since PUSH decrements esp before placing the operand on the stack, an LEA instruction (1) is inserted, adding 4 to the stack pointer to compensate for the deduction made by PUSH. esp thus points to the second free slot before PUSH (2), and directly to the pushed value after this instruction (4). At this point, the value pointed to by eax has been placed on the stack. The LEA that is inserted afterwards (5) causes esp to point to the next free slot (6), to be ready for the next stack operation. This works similarly with POP, but in this case, the LEA instruction decreases the stack pointer before and after it, as POP increases esp internally after it has retrieved the value from the stack (in contrast to PUSH, which decreases it before it places the value).

The CALL and RET instructions are more difficult, since control is transferred after the instruction, which means that the stack pointer cannot be adjusted directly afterwards. In the case of CALL, the pointer is adjusted (incremented) before and in the prologue of the called function. But this can cause problems if the function is called with a JMP instead of a CALL, and thus esp is decremented before a JMP in order to compensate for the adjustment in the prologue of the function. RET can be called with an operand that indicates how many bytes need to be cleaned up before returning (usually the function arguments). Salamat et al. [76] handle this case with a POP, LEA, JMP sequence, which they do not describe in detail, but Figure 4.2 outlines how this might work.

However, the layout of C/C++ structs, large data units, and arrays must be preserved (and not reversed internally), since they might be copied (say, with memcpy) to other memory segments (like the heap), and must thus remain compatible. This requires that information is available during compilation, and prevented the researchers from implementing a generic translation tool that would operate on executables without this information [76].

Some overhead results from running an executable with a reverse stack, since ad-
additional instructions are necessary. However, modern superscalar CPUs can execute multiple instructions in parallel (as can single-core machines), so they do not cause high performance penalties [76]. The benchmarks conducted by Salamat et al. show an average runtime overhead of 3%, with a 10% increase in code size.

This approach is effective in probabilistically detecting manipulations of both control and noncontrol data. However, it is limited to the stack and, under certain circumstances, a program might still be exploitable through a stack-based buffer overflow. See, for instance, the stack layout in Figure 4.3. On the left, a normal (for x86) downwards-growing stack is depicted, and on the right the reverse stack can be seen. In this example, a function pointer located before the buffer is involved. It is, in fact, likely that function pointers are located before the buffer, since this is more secure under normal circumstances (as the buffer cannot overflow into the function pointer). In a reverse stack, however, the function pointer is located after the buffer, and thus can be overridden. While it cannot be overwritten on a downwards-growing stack, the return address can be targeted instead. If the function pointer is called before returning, it does not matter which pointer is overwritten, as eventually both program variants will execute the (malicious) code specified by the pointer. This is a naïve example, in that most compilers add stack protection, such as canaries, during compilation, and this might prevent the overwriting of the return address. However, these protections are themselves probabilistic, and are not necessarily always applied. The example shows that reverse-stack execution with normal execution cannot by itself prevent all stack-based buffer overrun exploits, and that it needs to be combined with other protections if reliable security is to be achieved.

4.2.2 Address Space Partitioning

Cox et al. [21] and Cavallaro et al. [14] [10] both chose address-space partitioning as a diversification approach in the N-Variant and DPR system, respectively. Their approaches achieve similar results, but one is a kernel implementation and the other is a userland implementation, so there are significant differences. Since a userland implementation is preferable, this is what will be focused on in the following. The general assumption (unexpressed in the papers) is that a process does not use its entire address
space. Thus, the address space can be cut in half, with half assigned to one process and half to its replica or variant. The \texttt{.text}, \texttt{.data}, and \texttt{.bss} of ET\_EXEC objects can be relocated at link time by modifying gcc's linker script [21] [14] [10]. Since the heap follows \texttt{.bss}, its location can also be handled by the linker script [14]. Cox et al. try to ensure nonoverlapping address spaces between the variants by limiting the sizes of the address space [21], whereas Cavallaro et al. [15] map the dormant partition with \texttt{PROT\_NONE}. As discussed in Chapter 3.0.2, the latter is the preferable approach.

However, other segments of a process' address space, such as the stack and shared libraries, cannot be relocated at link time, and must be relocated at runtime. Cavallaro et al. [14] [10] thus modified the dynamic linker 1d-\texttt{linux\_so}, which loads shared objects at runtime. Their modifications enable the relocation of the stack and of dynamically linked shared objects at runtime.

A downside of address-space partitioning is that it may not be compatible with every program. A program that depends on its memory location, and which changes semantics as a consequence of different virtual addresses, will not be compatible. A program that makes a decision depending on the value of a pointer, or one that prints out its memory address [21] [14], would be examples of this. This would mean that the monitor would raise a false alarm, as it would assume that a program is being exploited once the variants diverge.

This approach is effective in deterministically detecting every full pointer overwrite, no matter in which segment of a process' address space it occurs. However, partial pointer overwrites, or noncontrol data attacks that do not modify a pointer, might not be detected. To improve detection, Cavallaro et al. apply address-space shifting to add a probabilistic detection for partial pointer overwrites. Partial pointer overwrites—which can lead to the execution of normally impossible paths—are easily achieved on little-endian systems, since the least significant byte is located first in virtual memory. This means that a buffer overflow that exceeds the buffer by one byte will overwrite the least significant byte of the adjacent multibyte value (e.g., a pointer, which is 4 bytes on a 32-bit system). Such overwrites could cause the program to skip some instructions or to execute code that differs from that intended by the programmer. Since both variants of the program would do this equally (as the modified least significant byte can be
made valid for both variants), the monitor cannot detect any divergence in behaviour and assumes that the variants are working correctly. Address-space shifting means that the address spaces of the variants are not only disjoint in regards to their address space, but also differ in their least significant byte. Thus, if the least significant byte of a pointer is overwritten, execution is redirected to a different location when the pointer is invoked. This is likely to change the semantics of the program, which will be detected by the monitor. However, the semantics need not necessarily change, which makes this protection merely probabilistic. The simplest example to think of is a program that uses NOPs (or equivalent instruction that do not change the program state). If one variant jumps to the (malicious) code intended by the attacker, while the other jumps to the NOPs prior to the (malicious) code, both programs will eventually execute the (malicious) code equally without detection. Other instructions are possible too, if they are below the monitor’s radar (which depends on the monitoring granularity—see Chapter 4.3). The following graphic illustrates such an exploit:

\[
\begin{array}{c|c|c}
\text{initial eip} & \text{P1} & \text{P2} \\
0x8048474 & \text{security-sensitive code} & \text{eip 0x8048478} \\
0x80484a0 & \text{NOP} & 0x80484a4 \\
0x80484a1 & \text{NOP} & 0x80484a5 \\
0x80484a2 & \text{XOR %eax, %eax} & 0x80484a6_eip \\
0x80484a4 & \text{XOR %ebx, %ebx} & 0x80484a8 \\
eip & 0x80484a6 & \text{XOR %ecx, %ecx} \\
0x80484a8 & \text{malicious code} & 0x80484ac \\
\end{array}
\]

Address-space partitioning and shifting do not in themselves add any runtime overhead, since no additional instructions are executed.

4.2.3 Address Space Layout Randomization

Address-Space Layout Randomization, as described in Chapter 2.5, randomizes the start address of memory regions in the process’ address space. This makes it harder for an attacker to find the memory address to which the execution flow needs to be redirected if the malicious operations are to be executed. As already explained, ASLR only offers adequate protection if the binary is compiled as a Position-Independent Executable (PIE), so that the location of the .text segment can be randomized, meaning that Return Oriented Programming (ROP) can be defeated probabilistically. Brute force attacks are theoretically feasible, especially on 32-bit systems, where the entropy is often no greater than 16 bits [82]. This approach, like all other probabilistic approaches, relies on keeping the randomization seed secret. An attacker could use information leaks to determine the randomization, and thus to predict memory addresses of particular instructions or data. When ASLR is used with a replicated execution system, the chances for a successful brute force attack can be decreased drastically, since the attacker needs to find instructions that perform the intended operations in both variants (which are randomized differently). Even with a leak of information, the attacker still needs to find an address in both variants that contains equivalent instructions performing the intended operation.

Position-Independent Executables are associated with some performance overhead, which is one reason why not all executables are compiled as PIEs [67]. According to the
benchmarks performed by Payer M. [67], the performance overhead of a sample set of PIEs (compiled with -03 -fPIE), compared with their non-PIE versions, can be up to 25.89%, and has an average of 10.12%.

4.2.4 Comparison of Diversification Approaches

Besides the approaches to memory diversification that have been previously explained, other forms of diversification exist as well. As the focus of this work is on memory diversification, these other approaches will only be covered briefly.

Besides address-space partitioning, Cox et al. presented the idea of instruction-set tagging for their N-Variant System [21]. This diversification transforms a binary and inserts a tag bit before each instruction (binary rewriting). The binaries of both variants receive different tags. Before execution of each tagged instruction, the instruction tag is checked; if it is valid for the variant that is trying to execute it, the tag is removed and the instruction is passed to the CPU for execution (software dynamic translation) [21]. Since I/O is synchronized, and thus both variants are fed the same input, an attacker must decide how to tag the instruction, and it is not possible to prepend tags to the instructions of the malicious code that are valid in both variants. Thus, upon code injection, the attack will be detected deterministically. However, since this method attempts to detect the execution of injected code, it cannot prevent an attack that makes use of existing code, such as return-into-lib or ROP attacks.

Another approach is instruction-set randomization, investigated by Kc et al. [52]. In their research, binaries are transformed to randomize and encrypt instructions using XOR. The key is stored in the ELF header, and not in the process' address space itself, and is thus protected. The researchers proposed direct processor support for de-randomization and decryption of the instructions, but also implemented an emulation in which the instructions are decrypted prior to being passed to the CPU for execution [52]. This method thwarts the execution of injected code but, like instruction-set tagging, is vulnerable to return-into-lib and ROP attacks. In contrast to the previously presented mechanisms of diversification, but similarly to the technique that will be presented next, this approach is merely probabilistic, as it is theoretically possible (though improbable) that a random sequence of bytes injected by an attacker would result in valid instructions. The approach is also vulnerable to information leaks, in which the attacker manages to read out some instructions. This is problematic, as an attacker could launch a brute-force or even a known-plaintext attack against the randomized instructions to retrieve the key. This key could then be used to randomize the injected malicious instructions and avoid detection. However, if this diversification is employed together with replicated execution, the attacker would also need to find a byte sequence that produces instructions that perform the intended task after decryption with the different keys of both variants.

Randomizing system calls is another probabilistic approach to preventing the execution of injected code containing system calls. Since a system call is the interface between a process and its environment, it can be assumed that no harm can be caused to the system without a system call being issued. Jiang et al. [50] present a diversification approach based on system call number randomization. In their implementation, system calls are encrypted, either at load time or offline prior to execution. The encryption
function takes into account a key, as well as the position of the system call in the process’ address space, which means that the same system call is randomized to a different value at each appearance in the binary. The key is stored in kernel space, and is thus protected from unauthorized access. The researchers performed derandomization at runtime in kernel space by modifying the system call dispatcher [50]. Jiang et al. combined this method with other diversifications to enhance protection, but in this comparison, the techniques are analysed individually and for use in a replicated execution environment. The attacks that this approach protects against, as well as the attacks it is vulnerable to, are very similar to those of instruction-set randomization. The disadvantage of system call randomization is that an attacker could, without being detected, perform arbitrary computations so long as no additional system calls are injected. In principle, system call randomization should perform much better than instruction-set randomization implemented in software, since the instructions do not need to be emulated, and additional computation only occurs on system calls.

See Table 4.1 for a direct comparison of all the approaches discussed. Note that none of the techniques can prevent DoS attacks, and all probabilistic approaches rely on keeping some sort of seed or key secret; they are thus vulnerable to information leak attacks (such as those that exploit format-string vulnerabilities).

<table>
<thead>
<tr>
<th>Diversification</th>
<th>Protection</th>
<th>Relab.</th>
<th>Vulnerable to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction-set tagging</td>
<td>Execution of injected instructions</td>
<td>Det</td>
<td>Noncontrol data attacks, return-into-lib/ROP Pointer</td>
</tr>
<tr>
<td>Address-space partitioning</td>
<td>Call of injected pointers</td>
<td>Det</td>
<td>LSB overwrites, nonpointer manipulation attacks</td>
</tr>
<tr>
<td>Reversing stack direction</td>
<td>Attacks on the stack</td>
<td>Prob</td>
<td>Attacks in other memory segments, information leaks</td>
</tr>
<tr>
<td>Instruction-set randomization</td>
<td>Performing intended operations with injected instructions</td>
<td>Prob</td>
<td>Noncontrol data attacks, return-into-lib/ROP, information leaks</td>
</tr>
<tr>
<td>System call randomization</td>
<td>Performing intended operations with injected system calls</td>
<td>Prob</td>
<td>Noncontrol data attacks, return-into-lib/ROP, information leaks</td>
</tr>
<tr>
<td>full ASLR</td>
<td>Redirection of execution flow to intended code</td>
<td>Prob</td>
<td>Noncontrol data attacks, information leaks</td>
</tr>
</tbody>
</table>
Since address-space partitioning promises deterministic prevention of all control data attacks, except for LSB overwrites, without much performance impact, this approach will be used in the further investigation and for implementing a prototype.

4.3 Monitoring and Synchronization

The three approaches to replication and diversification (N-Variant, MVEE, and DPR) agree that monitoring of the replicas at system call granularity is sufficient. More finely grained monitoring would come at the cost of considerable overhead, without much gain in security. This is because, in principle, a process should not be able to interact with the environment without executing a system call. There are, however, some exceptions to this, which are discussed in Chapter 4.5.

Both of the investigated user-mode implementations utilize the ptrace system call [14] [10] [75]. ptrace is usually used for debugging, and allows tracing the activity of another process. Its permits examining and modifying the tracee's address space and registers. Apart from single-stepping the tracee, it also allows the tracing of system calls only (ptrace's PTRACE_SYSCALL request), which is the granularity required. The PTRACE_SYSCALL request stops the traced process directly before a system call, and after it has executed a system call, and transfers control to the tracing process. This allows the tracer to inspect and modify the system call and its arguments [1]. However, ptrace does not allow a system call to be skipped. Since this is required in some cases, another system call that does not change the program's state must be executed [77]. Alternatively, an erroneous system call (for example, one with invalid arguments) could be executed.

Although ptrace is efficient for monitoring system calls, it is very inefficient for copying data between the tracer and tracee, as is required for I/O synchronization. The reason for this is that ptrace only allows 4 bytes of data to be copied at a time, and so multiple calls are required to copy more, each requiring two context switches (one to and one from the traced process).

The monitor has three functions: it executes both variants as its child processes, it checks if the variants are synchronized (that is, if they are issuing the same system calls with equivalent arguments), and it replicates input from one variant to the others so that they are kept synchronized under normal execution conditions (when not under attack). It also needs to conflate output and to terminate a variant if the other one becomes compromised—for example, if it receives a SIGSEGV signal or if the system calls differ.

4.3.1 Output Comparison by the Monitor

At first glance, it might seem that a comparison of system call parameters is not necessary, since address-space partitioning means that it should not be possible to compromise one variant without causing a segmentation fault in the other. This might indeed be the case for control data attacks, but consider the following scenario in which a noncontrol data attack targets file descriptors (v1 and v2 are the two variants): Both variants issue an open system call for the file f1, but v1 receives file descriptor 5 and v2 file descriptor 6 from the kernel. The file is still open when, later, both processes open another file.
f2. This time, v1 receives file descriptor 6 and v2 receives file descriptor 5. At this point, both variants have behaved the same, except that they have received different file descriptors—not normally an issue. If an attacker, however, manages to overwrite the file descriptor for the next write operation on f1 with the number 5 for both variants, then v1 will write to f1, but v2 will instead write to f2. Since the write to the file itself is performed by the kernel, the monitor would not detect the issue if it did not check for the equivalence of system call parameters. Thus, an attacker could manipulate open files without detection. To prevent this, the monitor needs to keep a mapping of file descriptors, in order to be able to check their equivalence.

4.3.2 Monitor–Variant Communication

Since the monitor runs in user-mode, it cannot access the variant’s memory and register states directly. The monitor can only do this through the ptrace system call, which provides the PTRACE_PEEKDATA and PTRACE_POKEDATA requests that can be used to modify the tracee’s memory. These methods only allow the transfer of 4 bytes per request, and many context switches are thus necessary to copy a large buffer. This limitation makes data transfers from monitor to variant, or vice versa, very slow.

The authors of MVEE [75] propose to instead use shared memory for monitor–variant communication. According to their measurements, ptrace is the most efficient method for buffers smaller than 40 bytes, but for all buffers larger, shared memory is much more efficient: “For a 128 KB buffer, shared memory is more than 900 times faster than ptrace” [75]. This means that performance can be increased significantly if shared memory is used in places of ptrace.

Salamat et al. [75] propose injecting code that performs the copying to and from shared memory. The locations of the buffers are provided by the registers, and the instruction pointer is manipulated to invoke the injected code. This code first attaches a shared memory block created by the monitor, and then performs the copying. A dummy system call at the end of the injected code transfers control back to the monitor.

However, shared memory does not come without security concerns. Arbitrary processes must be prevented from manipulating the contents of the shared memory. Salamat et al. [75] set the permissions of the shared objects so that only the user running the monitor can access them. Manipulating the shared memory thus requires that the system is already compromised, so that a malicious process can run with the monitor’s user identity. Additionally, each shared memory block has a key that processes need to provide, though such a key does not provide full protection, as it could be guessed or brute-forced. One variant cannot attach another variant’s shared memory block, since a system call would be required to do that, and this can be controlled by the monitor [75].

4.3.3 Protection of the Monitor

A compromise of the monitor itself, either by manipulating its address space or by feeding it with malicious input, would be a serious issue. Since the monitor runs as a separate process, its address space cannot be accessed directly by any variant, as to achieve this, a system call would have to be issued (e.g., ptrace)—but this can be caught by the monitor and handled appropriately (that is, blocked and the variants terminated). Also,
as with any security system, we need to assume that it works correctly and that it is
not possible to compromise it by feeding it malicious input which might cause a memory
error in the monitor itself.

4.4 System Calls

4.4.1 Categorization

Linux 3.8 provides 350 system calls that all need to be handled properly by the monitor.
The authors of DPR [14] [10] and of N-Variant [21] categorize system calls in similar ways,
and distinguish three different types. In this report, the names of the DPR categories
are used.

Simulated system calls are compared for equivalence and executed by one variant
only. The result is then copied into the address space of the other variant. An
equivalent example of such a system call is read. Once a variant stops at the system call, it is
checked whether the other one has already tried to issue this call. If it has not, the
variant waits. Otherwise, the monitor checks whether the variants want to execute
the same system call—that is, if the system call numbers are the same and the
file descriptors equivalent. If this is the case, one variant is allowed to execute the
call. Immediately after the system call, the result is copied from the executing
variant to the other, and both can proceed with execution. Inconsistencies could
result if both variants executed the call directly, as process scheduling might lead
them to execute it at slightly different times. If the file they are reading from
changes after one variant has executed the system call, but before the other has,
they would potentially read different data, and thus become unsynchronized even
though the process is not under attack. This would result in a false positive and
in the termination of the variants.

Executed system calls need to be executed by both variants. This may be either
because the result is immutable (such as in the case of the uname system call, which
retrieves static OS information like version and hardware identifiers) or because the
system call modifies kernel data structures (such as the open system call, which
creates the kernel data structures for the file handle that is used in subsequent
system calls that need to be executed by each variant directly, such as mmap).

Carefully treated system calls are those that need special treatment, because they
could otherwise render some of our assumptions for address-space partitioning
and isolation of processes void. Such system calls include mmap and mprotect.
mmap creates a new (file) mapping in the virtual address space of the calling
process [1]. This could be used to map shared libraries, but also for inter-
process communication. If the variants map the same file into their address
spaces (MAP_ANONYMOUS flag not set) and propagate changes back to the file
(the MAP_SHARED and PROT_WRITE flags set), this would create a communication
channel between the variants and could harm the security of the system or cause
inconsistencies, thus leading to false positives. In other cases (MAP_ANONYMOUS, or
PROT_WRITE not set), mmap can be executed by both variants without any further

31
special treatment, as long as the mapping is within the variant’s address-space partition. Cavallaro et al. [14] [10] proposed a method to handle nonanonymous shared mapping with write permissions by utilizing protection on pages. Chapter 4.4.3 shows the relevance of such uses of \texttt{mmap} and Chapter 4.5 discusses shared memory in more detail.

MVEE [75] handles system calls differently. In general, system calls are executed by the monitor and the results are replicated into both replicas. Where this is not possible, the system call is executed by the variants. System calls that observe some information from the variant processes should be executed by at least one of the variants in order to return the correct information—rather than being executed by the monitor, which might run in an environment with different privileges, resources, and so on. Because of this, DPR’s approach seems to be more consistent, as the monitor rarely needs to execute a system call on behalf of the variants; so system calls can thus almost always be executed by at least one variant. An example of a sequence of system calls that requires special treatment when using MVEE’s approach, but can be handled normally with DPR’s approach, is the following: Consider the \texttt{open} system call. This system call modifies in-kernel data structures to create the file handle. The authors of MVEE decided to allow the monitor to execute \texttt{open} and to let it handle all file access [75]. However, a subsequent \texttt{mmap} system call requires a file descriptor as an argument. \texttt{mmap} must be executed by the variants themselves, as it can hardly be simulated by the monitor with acceptable performance. A file descriptor is only valid in the context of the process that created it. For \texttt{mmap}, this might mean that code needs to be injected to execute the \texttt{open} system call again in the context of the variant, having been previously executed by the monitor. The authors of MVEE resolved this problem by having the variants execute the \texttt{open} and \texttt{read} system calls directly, as long as they do not require write permissions. This can cause race-conditions, because the variants are unlikely to be executed in parallel and a file might be altered after one variant has read it, but before the second has. This can lead to inconsistencies and thus to false positives [75] [77]. DPR’s approach instead allows both variants to always execute the \texttt{open} system call, so that both have the necessary in-kernel data structures. On the other hand, as described above, only one variant executes system calls like \texttt{read} and \texttt{write}. There are system calls that are handled more simply using MVEE’s method, but overall DPR’s method seems to be more consistent. To avoid the variants diverging due to different file descriptor values, both variants can be fed the same file descriptor by the monitor (a virtual file descriptor), which is replaced by the real file descriptor (stored by the kernel) when a system call is issued.

From a performance perspective, the approaches of MVEE and DPR are similar. For instance, the \texttt{read} system call, when executed by the monitor, requires one execution of the system call by the monitor, skipping the system call by all the variants, and replicating the results to all variants. When one variant executes the system call, this execution must take place and the other variants are required to skip the call. The result must then be read from the variant that executed the \texttt{read} call and copied into the other variant. So, in total, there is one system call execution less in DPR’s approach, because rather than skipping the system call in all variants, one of these calls is used to perform the actual operation, and is thus not wasted. This is why it was decided to use the latter method for the author’s prototype implementation.
During the research conducted, all Linux system calls were analysed and summarized in a list together with their parameters and return values. This list also includes options for how the system calls are handled by the monitor and a number of potential issues. The list is available from the project’s code repository; information on how to access this repository can be found in the Appendix A.2.

4.4.2 Handling of System Calls

Some system calls are less “natural” to handle than others. While many system calls perform only a single operation, others perform one of many different operations depending on the arguments. One such system call is `socketcall` [1]. All the different `libc` socket functions are implemented through this single system call. The first argument specifies which operation to perform, taking on any out of 18 possible values (Linux 3.8.0). Examples include `SYS_SOCKET`, `SYS_BIND`, `SYS_CONNECT`, `SYS_LISTEN`, `SYS_SEND`, and `SYS_RECV`. Another system call that is very difficult to handle is `ioctl` [1], which modifies the parameters of a device. Its operation and arguments are completely dependent on the device driver. It is not even defined whether the arguments are input or output parameters. Without deeper knowledge about the device driver, it cannot be determined what operation the system call is going to perform. For the monitor, this means that it is hard to determine whether (and how) values should be checked or replicated. This makes it difficult for our monitor prototype to support the call. But, as our statistical analysis in Chapter 4.4.3 shows, this system call seems to be used rarely, and was not executed even once during our test cases described in Chapter 4.4.3.

4.4.3 Statistics

To learn more about the relevance of certain system calls and how frequently they are used in practice, some statistical information has been generated by the author. Some common desktop applications were traced using `strace`, which is a Linux debugging utility that makes use of `ptrace` to investigate system calls, their arguments, and results. A combination of Linux tools, such as `sed`, `sort`, `uniq`, `grep`, and `wc`, was used to count and rank system calls from `strace`’s output. See the Appendix A.1 for the scripts used.

<table>
<thead>
<tr>
<th>Rank</th>
<th>System call</th>
<th>#</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clock_gettime</td>
<td>81,086</td>
<td>30.60</td>
</tr>
<tr>
<td>2</td>
<td>gettimeofday</td>
<td>69,900</td>
<td>26.38</td>
</tr>
<tr>
<td>3</td>
<td>futex</td>
<td>42,008</td>
<td>15.85</td>
</tr>
<tr>
<td>4</td>
<td>recv</td>
<td>17,622</td>
<td>6.65</td>
</tr>
<tr>
<td>5</td>
<td>poll</td>
<td>15,108</td>
<td>5.70</td>
</tr>
<tr>
<td>6</td>
<td>madvise</td>
<td>4916</td>
<td>1.86</td>
</tr>
<tr>
<td>7</td>
<td>writev</td>
<td>4769</td>
<td>1.80</td>
</tr>
<tr>
<td>8</td>
<td>read</td>
<td>4735</td>
<td>1.79</td>
</tr>
<tr>
<td>9</td>
<td>write</td>
<td>4399</td>
<td>1.66</td>
</tr>
<tr>
<td>10</td>
<td>stat64</td>
<td>2691</td>
<td>1.02</td>
</tr>
</tbody>
</table>

33
Browsing two websites using Mozilla Firefox 22.0 caused the execution of 105 different system calls, with a total of 265,001 system calls made. The ten most frequently executed are shown in Table 4.2; they account for more than 93% of system calls executed. As we can see, date and time (clock_gettime and gettimeofday), network and file I/O (recv, poll, write, read, write, and stat64), handling of concurrency (futex), and performance optimizations of paging (madvise) are the most heavily utilized system calls in this kind of application.

Another system call that is of special interest is mmap2. mmap2 is essentially the same as mmap, but it supports mapping of larger files [1]. This was called 1,465 times, which makes it the 15th most frequently executed system call. Four of these calls had the MAP_SHARED flag set together with PROT_WRITE permission, which must be treated specially in a replicated and diversified execution system (further described in Chapter 4.5). Firefox forked its process using the clone system call 47 times, 41 times of which it shared its address space with the child (CLONE_VM flag set).

<table>
<thead>
<tr>
<th>Rank</th>
<th>System call</th>
<th>#</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>read</td>
<td>50,911</td>
<td>26.30</td>
</tr>
<tr>
<td>2</td>
<td>fstat64</td>
<td>42,409</td>
<td>21.91</td>
</tr>
<tr>
<td>3</td>
<td>write</td>
<td>33,521</td>
<td>17.32</td>
</tr>
<tr>
<td>4</td>
<td>close</td>
<td>20,618</td>
<td>10.65</td>
</tr>
<tr>
<td>5</td>
<td>openat</td>
<td>20,595</td>
<td>10.64</td>
</tr>
<tr>
<td>6</td>
<td>fstatat64</td>
<td>20,595</td>
<td>10.64</td>
</tr>
<tr>
<td>7</td>
<td>fcntl64</td>
<td>2407</td>
<td>1.24</td>
</tr>
<tr>
<td>8</td>
<td>getdents64</td>
<td>2404</td>
<td>1.24</td>
</tr>
<tr>
<td>9</td>
<td>open</td>
<td>36</td>
<td>0.02</td>
</tr>
<tr>
<td>10</td>
<td>mmap2</td>
<td>26</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Archiving the source code directory tree of the Linux 2.6.16 kernel using GNU tar with “-cf” resulted in 22 different system calls and a total of 193,580 system calls being executed. Table 4.3 shows the system calls that were issued most often. As expected, almost all of these are file or file-system operations. Of the 26 mmap2 executions, zero requested a nonanonymous shared mapping with write permissions. GNU tar does not execute any call from the fork family of system calls, and thus executes in a single thread.

A system call trace of the Apache 2.2.22 processes while performing benchmarking using Apache’s HTTP server benchmarking tool ab, with 10 concurrent requests and 1000 requests in total, showed that 45 different system calls were executed. In total, 563,403 system calls were executed. As can be seen in Table 4.4, most of the operations were performed on the file system. Out of the 28,190 executions of mmap, none requested a nonanonymous shared mapping with write permissions. The Apache processes issued 32 calls to clone, none of which requested a shared address space with the child.

34
Table 4.4: Httpd—System calls

<table>
<thead>
<tr>
<th>Rank</th>
<th>System call</th>
<th>#</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lstat64</td>
<td>128,425</td>
<td>22.79</td>
</tr>
<tr>
<td>2</td>
<td>time</td>
<td>89,192</td>
<td>15.83</td>
</tr>
<tr>
<td>3</td>
<td>close</td>
<td>45,284</td>
<td>8.08</td>
</tr>
<tr>
<td>4</td>
<td>open</td>
<td>37,528</td>
<td>6.66</td>
</tr>
<tr>
<td>5</td>
<td>read</td>
<td>34,057</td>
<td>6.05</td>
</tr>
<tr>
<td>6</td>
<td>lstat64</td>
<td>30,865</td>
<td>5.48</td>
</tr>
<tr>
<td>7</td>
<td>munmap</td>
<td>29,810</td>
<td>5.29</td>
</tr>
<tr>
<td>8</td>
<td>mmap2</td>
<td>28,190</td>
<td>5.00</td>
</tr>
<tr>
<td>9</td>
<td>access</td>
<td>27,718</td>
<td>4.90</td>
</tr>
<tr>
<td>10</td>
<td>gettimeofday</td>
<td>25,653</td>
<td>4.55</td>
</tr>
</tbody>
</table>

4.5 Nondeterminism Issues

4.5.1 Threads

Threads (or lightweight processes) can cause some nondeterminism issues when used by processes that are being traced by the monitor at system call level. Since the order in which threads execute depends on the scheduler and cannot be predicted easily, the two variants might execute some operations in a different order. This means that different system calls would be executed by the variants, which would cause the monitor to raise a false positive and to shut down the processes. Since this deviation in behaviour is not malicious, but it is rather how threads are designed, a solution to this issue needs to be found. When a `fork` occurs, the monitor needs to synchronize the child processes or threads separately from their parents. In the following illustration, the monitor process will be designated m1-1 and the variants v1-1 and v2-1. Their children will be called m1-2, v1-2, and v2-2 respectively. Salamat et al. [75] in their research describe how, when variants `fork`, the monitor also needs to `fork`. m1-1 then needs to monitor and synchronize the variants' children, v1-2 and v2-2, whereas m1-1 continues its work monitoring v1-1 and v2-1. This approach should work correctly when the children are separate processes or threads that do not run in the same memory space as their parents (the CLONE_VM flag is not set in the system call).

However, the researchers have not indicated that this approach is not reliable when the threads share their address space with their parents. Imagine that the variant's children v1-2 and v2-2 modify a global variable gvar, and the parents v1-1 and v2-1 make a decision `dec` depending on this variable. This means that, depending on gvar, different code and different system calls will be executed. If `v2-2` is scheduled differently from `v1-2` so that it is able to modify the global variable before its parent makes the decision `dec`, but `v1-2` cannot, then `v1-1` and `v2-1` will diverge in behaviour and the monitor will raise a false positive. Figure 4.4 illustrates this.

To avoid such issues, more finely grained synchronization or synchronous scheduling is necessary. If it can be ensured that `v1-2` and `v2-2` receive exactly the same CPU time between two successive executions of their parents, then the described issue can be avoided. Also, if the monitor were to trace the variants with a finer granularity, it
could handle such situations appropriately. One potential solution is to let the monitor trace the variants at instruction level. This, however, would mean that multiple context switches per instruction would be necessary, which is not tolerable from a performance perspective. Another approach would be to mark the regions that contain global variables (such as `.data` and `.bss`) as inaccessible (mapped with PROT_NONE). This would give control back to the monitor on each attempt to access a global variable, and the monitor could then synchronize its modifications. This approach will still have a considerable performance impact on programs that make heavy use of global variables.

4.5.2 Signals

Signals are problematic, as they are not deterministic by nature and so are practically impossible to predict. The “time” (that is, the state of execution of a process) at which a signal is received, however, can influence the behaviour of the application. Imagine that variant v1 issues the system call s1, but v2 receives a signal that changes its behaviour to issue system call s2 instead. Only now v1 receives the signal. The monitor will compare s1 with s2 and, since they are not equal, the variants are unsynchronized and a false positive is raised. Since the monitor can trace the variants and thus control signal delivery, this control could be made use of. Cavallaro et al. [14] [10] proposed delaying the delivery of signals to a synchronization point (system call), in order to ensure perfect synchronization. This method solves the issue of false positives. It also, however, means that CPU bound processes, which perform a lot of computation before issuing a system call, might not be able to make use of the signal [14] [10]. This might not be acceptable for some types of signals [5].

Salamat et al. [75] [5] improve on the above approach with their assumption that signal delivery to all variants at any time between the same two system calls is sufficiently fine-grained. A case where this assumption might not hold is described later in this chapter. The term “system call frame”, used in the following paragraph, means all stages of execution of a process between two successive system calls. The proposed method is based on a majority voting system. If all variants have received a signal within the same system call frame, then the monitor forwards the signals to the variants.
immediately (even before the next system call). This avoids long delays for CPU-bound processes. If at least half of the variants have received a signal before the next system call, the signal is delivered to them. But for the variants that have not received the signal, the next system call is skipped, and these are forced to wait until the signal is delivered. If a minority have received a signal before the next system call, the signal is delayed until either a synchronization point is reached where a majority has received the signal, or until all variants have received the signal. This ensures that signals are delivered to all variants within the same system call frame. Some more complex cases involving more than two system call frames exist, but these can be solved by combining the techniques mentioned above [75] [5].

This solution is not perfect, however, since it has the potential to change the behaviour of the program in cases such as the following: Imagine that a program runs a loop that modifies some variables on each iteration. The program’s signal handler could use one of these variables in decisions, which makes its behaviour dependent on the time at which the signal is received. Since the signal is delayed by the monitor, the behaviour of the program is altered, as compared to execution without the monitor [14] [10]. But such cases are assumed to be rare, as the nondeterminism of signals itself makes signals an unnatural basis for decision.

4.5.3 Shared Memory

Another issue is shared memory, since it allows the variants to communicate with their environment without issuing a system call. This means that the monitor loses control and could potentially be bypassed. Shared memory also opens a communication channel between variants, which is not intended as it could change their behaviour. The variants should be synchronized so that they cannot see that they are being executed in a replicated and diversified execution system; they should behave exactly as they would when executed normally. Imagine that variant v1 maps some shared memory backed by a file into its address space, and that variant v2 maps the same file. Now v1 makes a decision based on some variable in the shared memory. v2 does the same, but before it can execute, the mapped file is changed (by v1 or by an external process). This could mean that the opposite decision is made by v2 and that the variants diverge in behaviour. Retrieving this variable does not require any system calls, and so the monitor cannot synchronize the input from the mapped file. Shared memory is generally used by a single process and its child processes, and the involvement of external processes is rare, so the following discussion will focus on solutions to the problem on the assumption that the shared memory is only used by a variant and its child processes. A potential solution to the problem of external processes becoming involved has been suggested by Cavallaro et al. [14], and is described later in this chapter. Another issue that could arise is that if inconsistent data when both variants write to the shared resource.

The researchers behind MVEE [75] seem to make the same assumption about the noninvolvement of external processes: they always allow MAP_ANONYMOUS and MAP_PRIVATE, allow MAP_SHARED with read-only permission. This means that programs using MAP_SHARED with PROT_WRITE are not compatible with the system, which the authors argue "does not seem to be a significant limitation for most applications" [75]. Of the tests performed with Mozilla Firefox, GNU tar, and Apache httpd (described in Chapter 4.4.3),
only Firefox requested shared memory with MAP_SHARED and PROT_WRITE. Cavallaro et al. [14] [10] present a mechanism that allows MAP_SHARED mappings with PROT_WRITE permissions. One proposal the researchers make is to replicate the shared object, so that each variant has its own copy. This, however, could result in significant overhead if the shared objects used are large. Another approach is to change MAP_SHARED mappings to MAP_PRIVATE ones. This would have the downside that modifications to the data would not be propagated back to the shared object, which would change the behaviour of the processes. Thus, the researchers propose flushing out the modifications upon unmap of the shared memory region. This means that each variant would work on its own local copy of the shared object and that, once they unmap the area, the monitor would write back the modifications (only once) [14] [10]. This becomes problematic when child processes are involved, however, as they might access the shared memory at the same time as the parent. Imagine a case where the parent v1 maps a shared object into its address space. The mapping is altered from being MAP_SHARED to being MAPPRIVATE, which means that the parent works on its own local copy. Now the child v1-1 maps the same file into its address space and makes modifications. At this point, each of the processes v1 and v1-1 has its own (different) local copy of the shared object, but cannot see the changes of the other. This implies some sort of race condition. If v1 unmaps the memory first, its changes are flushed back to the object. However, when v1-1 unmaps the memory, its modifications are written back to the shared object, overwriting the modifications that v1 previously made. If the processes were scheduled differently, it could happen that v1’s modifications would persist.

If external processes were involved, even MAP_PRIVATE could not permitted without special handling: it works with copy-on-write, and thus reads from the original source until the shared memory region is written to, which could lead to inconsistencies similar to those described earlier in this section. Even if the processes were to write to the memory region at the beginning, and thus enforce copy-on-write, this would mean that the modifications made by external processes could not be seen by the variants, as would be the case under normal execution. A solution to the issue of shared memory with external processes was outlined by Cavallaro et al. [14] [10]. Their solution employs more finely grained synchronization on the part of the monitor. The idea is to map the shared area without permissions. Hence, a page fault will be issued by the CPU upon access of the shared memory region, and this can be caught by the monitor. The monitor can then allow access to one process, skip it for the other process, and replicate the result. This allows control that is fine-grained enough to handle the issue, but causes considerable performance overhead as multiple context switches are necessary for each access to the shared memory.

4.5.4 Other Issues

Operations producing different results for the diversified variants, and which cannot be synchronized by the monitor, are an issue, as they can lead to divergence of the variants and thus to false positives. For instance, if a program makes a decision based on an address, the two variants might execute different branches of the code. Another issue is instructions such as RDTSC on x86, which returns the current value of the Time Stamp Counter register. This register contains the number of clock cycles since the last reset,
and can be accessed using the RDTS instruction without a system call. It can thus not be synchronized with monitoring at system call granularity. More finely grained monitoring, such as at the instruction level, would allow synchronization of such operations, but would introduce additional overhead. Since this overhead is not likely to be acceptable in most environments, there is not much that can be done and applications that use this instruction for decision-making: they are not compatible with synchronization at system call granularity.
This section is intended to outline potential ways of implementing a multivariant execution system, and builds on the findings of Chapter 4. The objective is the implementation of a proof-of-concept prototype that can be used for further research.

5.1 Diversification

As address-space partitioning was chosen as the diversification approach in this implementation, two major issues need to be solved: The variant that uses the lower half of the user-mode address-space needs to have its stack moved into its partition (the stack is usually located at the end of the address-space). The other variant needs to have all its memory segments moved into the upper half. Segments that need to be relocated include .text, .data, and .bss. See Figure 3.2 for an illustration of address-space partitioning.

5.1.1 Moving the Stack

Since the stack is a memory region that is not present in the executable, but is rather mapped by the kernel at the start-up of the process, it is not possible to move the stack by modifying the executable. In a user-mode implementation, where the kernel cannot be modified, this means that the stack needs to be moved at the very beginning of program execution. This was implemented with a statically linked function called move_stack that performs all the operations necessary to move the stack to another address. This function is invoked at the beginning of execution by modifying the executable’s entry point and changing it to move_stack. This can be done using gcc’s -e option:

```bash
  gcc -e move_stack -o executable
  main.c
```

move_stack was implemented in C with some inline assembly. Pseudocode for the function that will be referred to during the following discussion can be found in Listing 5.1. The original source code for the move_stack function is located in the project’s code repository (see the Appendix A.2).

The move_stack function first saves the state of the edx register (9), which is one
of the general-purpose registers used to store the location of initialization code for the
dynamic linker. Since this register will be overwritten in the course of moving the stack, it needs to be restored at the end of the function. To determine the current top of the stack, the esp register is examined. The bottom of the stack is fixed at 0xc0000000 on Linux x86 with ASLR disabled. Since our approach to replicated and diversified execution is, in principle, stronger than ASLR (being secretless and deterministic), disabling it should not be a problem if all potentially vulnerable and exposed applications are run in our security environment (that is, as replicated, diversified, and monitored).

Next, the new stack area can be mapped into the process’ address space (19). This is implemented using the mmap system call, which maps a region with the same size as the original stack to the appropriate address in the lower partition of the address space. The new stack is allowed to be read from and written to. Its flags are MAP_ANONYMOUS, MAP_PRIVATE, MAP_GROWSDOWN, MAP_STACK, indicating that the mapping is not backed by a file, and that changes should thus not be written back to it. It also indicates that the memory region is a stack which grows towards lower memory addresses. If the mapping is successful, the old stack’s content is copied to the new stack (25) using the my_memcpy function—a slightly modified version of libc’s memcpy function. At this point, the ebp and esp registers can be changed to the new stack (28,29). From here on, the new stack is used.

However, pointers on the stack (argv, envp, and auxv) are copied without modification, and thus still point to addresses on the original stack. These pointers need to be modified (34), which is done by subtracting the difference between the location of the old and the new stack from the pointers. Additionally, gcc compiled the function as expecting a return address on the stack, which will not be there if the function has been executed at the entry point to the executable. This means that the variables argc and argv will not have been compiled correctly and reference a memory address 4 bytes (the size of the return address) higher than they should. This also needs to be adjusted.

The call to munmap unmaps the old stack from the process’ address space. This means that every further reference to this region would result in a segmentation fault. To provide additional security, the partition not assigned to the variant (that is, the upper half of the user-mode address space) is mapped without privileges (39). Since a system call is necessary to map a new region into the address space, which can be prevented by the monitor if it is not within the variant’s partition, this code is not absolutely necessary.

Next (41), a dummy system call is executed. This has the system call number 500 in our implementation, and lets the monitor know that the stack has been moved and that all successive system calls must be equivalent to the calls of the other variant (whose stack is in the original position). The variant that uses the upper half of the address space also has its entry point changed to a function that simply calls the dummy system call.

Towards the end of the function, the edx register that was backed up at the beginning is restored, and esp is adjusted to delete all the local variables created during the execution of this function so that later code can work from a fresh stack, as it expects. At the end of move_stack, a jmp instruction redirects the execution flow back to the original entry point. Another option would be to call main instead of jumping to start, but this would skip some libc initializations.
Listing 5.1: The move_stack function

```c
#define ORIG_BOTTOM 0xc0000000
#define REQ BOTTOM 0x68048000

void move_stack (int argc, int argv[])
{
    void orig_bottom = (void )ORIG BOTTOM;
    void req bottom = (void )REQ BOTTOM;

    Save the content of the register state
    
    / examine esp and ebp to know current stack /
    asm ("movl,esp-w %0;" : "=r" (orig esp));
    asm ("movl,ebp-w %0;" : "=r" (orig ebp));

    / Calculate the new top of the stack /
    req top = req bottom - (orig_bottom - orig esp);

    / map new stack /
    new_top = mmap(req_top, orig_bottom - orig esp,
                   PROT_READ|PROT_WRITE, .......);

    / if mapping was successful /
    if (new_top != (void ) - 1){
        / copy the content of the old stack to the new stack /
        my_memcpy(new_top, orig esp, (orig_bottom - orig esp));

        / set esp and ebp to new stack /
        asm ("movl,%0,esp;" : "r" (new top));
        asm ("movl,%0,ebp;" : "r" (new top +
                       (orig ebp - orig esp)));

        / -- FROM HERE ON THE NEW STACK IS USED -- /

        Adjust argv, envp and auxv

        / unmap the old stack /
        munmap(aligned_orig esp, (orig_bottom - orig esp));

        Map dormant address space partition without privileges

        Call dummy system call

        Restore original state (registers + stack)

        / jump to original entry point _start /
        asm ("jmp _start;" );
    }
}
```
5.1.2 Modified Linker Script

The last section mainly discussed the operations that are necessary for the variant that uses the lower half of the address space. This section explains how segments normally mapped to lower memory addresses are here remapped to memory addresses in the upper address-space partition. While the stack is mapped at runtime, and thus cannot be statically relocated, this is not the case for other memory segments. Memory segments in the lower memory partition are present in the executable and, by modifying the linker script, they can be relocated at link time. Fortunately, modifying one line in the default linker script is enough to remap all sections in the ELF file. The adapted line can be seen in Listing 5.2.

Listing 5.2: Modifications to the linker script

```c
1 PROVIDE ( executable start =
2   SEGMENT_START("text-segment", 0x68048000));
3   = SEGMENT_START("text-segment", 0x68048000) + SIZEOF_HEADERS;
```

The PROVIDE directive allows a symbol to be defined [69]. SEGMENT_START returns the base address of the specified segment. If a static value for the segment is passed to the linker as an argument, this value will be returned, otherwise the default value is provided as second parameter. In our case, this linker script sets the symbol _executable_start and the location counter (".") to 0x68048000. Modifying the location counter has the effect that all subsequently specified sections will be marked for mapping to memory locations greater than the location counter in the ELF file. Since the location counter is set to the start of the second partition, all memory regions of the process running this program will be mapped into this partition.

5.2 Monitoring

The monitor is implemented in C++ and utilizes the ptrace system call to trace the variant processes. To synchronize the variants at system call level, each system call has its own handler. The implementation at present supports the 20 system calls required by GNU tar to create an archive. Many system calls can be handled similar, and thus the implementation of additional system calls should not be an issue.

5.2.1 Code Structure

The two main classes are Variant and Monitor. An instance of Variant represents a single variant and handles its process. Its attributes store the process’ register state, its parent process, the executable file loaded into the process, the arguments of the program, and the mappings for associating virtual values (modified by the monitor) to real values (the ones the kernel has stored). Such mappings exist for process IDs and file descriptors. They are necessary because, as mentioned in Chapter 4.4.1, both variants receive the same values for identifiers in order to avoid divergence. These virtual values
are replaced with the original ones (the ones the kernel knows about) before any system call that uses them is executed. The virtual values also play a role in security, by helping to avoid the issues described in Chapter 4.3.1. The Variant class contains methods for starting the process, for initializing ptrace, for waiting for (and stopping at) the next system call, for executing or skipping a system call, and for modifying a process’ state (registers or memory). An instance of Variant can be used to run any executable and to monitor it with ptrace at system call level. This means that it is generic for any diversification method.

The Monitor class contains a list of the Variant instances it is intended to monitor. The monitor was also written generically, and thus it can be used with any diversification approach and with a variable number of variants. An instance of Monitor synchronizes its variants at the system call level, which means that it lets them continue until the next system call, compares and possibly adapts them and their parameters, handles the system call, and continues to the next one. How a system call is handled depends on the specific system call requested, with each having its own handler function that is also part of the monitor class. An example of the handler function for the read system call is shown in Listing 5.3. If the variants diverge in behaviour (for example, if they make different system calls, pass different parameters, catch different signals, or are in different process states), then all variants are shut down. Since some system calls need to be handled differently depending on the diversification used, their handler functions need to be adapted accordingly. This can be done by creating a child class of Monitor that will overwrite the handler functions in need of modification. Other parts of the Monitor class can be left unchanged.

5.2.2 Ptrace

ptrace is used to trace the variant processes and for monitor–variant communication. The ptrace requests used are PTRACE_ TRACEME, PTRACE_ SETOPTIONS, PTRACE_ SYSCALL, PTRACE_ PEEKDATA, PTRACE_ POKEDATA, PTRACE_ GETREGS, and PTRACE_ SETREGS.

When the monitor forks the variants, they execute a PTRACE_ TRACEME request prior to executing the execve system call that is used to load the variants into the address space. This ptrace request indicates that the process wants to be traced, and causes it to wait for the tracer on delivery of the next signal (which occurs at the next exec system call, at the latest). PTRACE_ SETOPTIONS is used to set the PTRACE_ Q_TRACE_SYSGOOD option, which allows the future stops of the tracee (variant) that are due to system calls to be distinguished from those stops not caused by a system call. The PTRACE_ SYSCALL request allows the variant to continue until it issues the next system call. PTRACE_ PEEKDATA and PTRACE_ POKEDATA are used to retrieve or modify the tracee’s memory, and PTRACE_ GETREGS and PTRACE_ SETREGS allow the contents of its registers to be inspected and changed [1].

5.3 Directions for Further Development

The implementation, in its current state, demonstrates the concept and outlines ways of implementing the described techniques. It is nonetheless a prototype state, and many
Listing 5.3: System call handler for read

```c
/ System call #3 /
/ variant[0] executes, all other variants skip system call and
result is copied from variant[0] to all other variants /
int Monitor::handle_read(){
  / return value /
  int ret = 0;

  char buf;

  / variant[0] executes system call /
  variants[0].execute_syscall();

  / all other variants skip system call /
  for(unsigned int i = 1; i < variants.size(); i++){
    / skip system call /
    variants[i].skip_syscall();
  }

  / if system call of variant[0] was successful /
  if(variants[0].get_regs_cache()->eax > 0){
    / get result from variant[0] /
    buf = new char[variants[0].get_regs_cache()->edx];
    if(variants[0].read_memory(
      (char *)variants[0].get_regs_cache()->ecx, buf,
      variants[0].get_regs_cache()->eax) < 0)
      return -50;
  }

  / replicate data read from variant[0] to other variants /
  for(unsigned int i = 1; i < variants.size(); i++){
    / if system call of variant[0] was successful /
    if(variants[0].get_regs_cache()->eax > 0){
      / write the data from variant[0] to variant[i],
      the return value of variant[0] (eax)
      indicates how many bytes were read /
      if((ret = variants[i].write_memory(
        buf, (char *)variants[i].get_regs_cache()->ecx,
        variants[i].get_regs_cache()->eax)) < 0)
        return ret;
    }
  
  variants[i].set_reg_cache(
    (unsigned int)(variants[0].get_regs_cache() +
    offsetof(user_regs_struct, eax)),
    offsetof(user_regs_struct, eax));

  return 1;
}
```
functions necessary to use it productively are still missing. Such missing functionality includes support for child processes (or threads), which requires a new monitor thread to be created in order to watch and synchronize the child processes. Also, many system call handlers are yet to be implemented. Monitor–variant communication is currently done using ptrace, which is, as mentioned in Chapter 3.0.2, a very inefficient method from a performance perspective. As proposed by Salamat et al. [75] and explained in Chapter 4.3.2, shared memory could be used for monitor–variant communication to increase performance.

Currently, dynamically linked shared objects are not mapped according to the memory partitioning technique. This means that if static linking is not used exclusively, vulnerabilities will inevitably be introduced, as dynamically linked shared objects do not comply with memory partitioning. Since these mappings provide the attacker with addresses that are valid in both variants, the attacker could return into one of these memory locations with both variants, in order to perform malicious operations. It is possible that such malicious operations are performed with equivalent system calls in both variants, in which case the exploitation could not be detected by the monitor. To resolve this issue, the dynamic linker itself needs to be relocated. Additionally, all the shared objects that it maps will need to be located in the appropriate memory partition. This could be done by modifying the dynamic linker [14] [10].

Moving the stack at runtime in user-mode also causes issues with the dynamic linker in the implementation’s current state, since it already runs before control is handed over to the program, and thus before the stack can be moved. During this execution, some global variables are initialized. After the stack has been moved, these variables reference incorrect addresses. One potential solution to this is to include the dynamic linker’s header files in the move_stack static library, so as to be able to adjust the global variables accordingly. Another possible solution is to modify the dynamic linker itself.
It is always a relief to believe what is pleasant, but it is more important to believe what is true

Hilaire Belloc (1870 - 1953)

Proof of Concept

To demonstrate the concept, a simple example application containing a stack-based buffer overflow vulnerability was created. The program simply greets the user by printing "Hello" followed by the user’s name, which is passed as an argument. Apart from the missing check if an argument was provided or not, the use of the unsafe strcpy function is what makes the program vulnerable: it copies the characters of a string into a buffer until a NULL-byte occurs, but does not check the target-buffer’s size. If the string is longer than the buffer, adjacent memory locations will be overwritten.

Listing 6.1: A Vulnerable Program

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char argv) {
    greetuser(argv[1]);
    _exit(0);
}

void greetuser(char argv){
    char name[100];
    strcpy(name, argv);
    printf("Hello%s\n", name);
}
```

To see the effect of the studied technique on its own, gcc’s stack protector were disabled at compilation (using the -fno-stack-protector flag), as can be seen in Listing 6.2. ASLR was disabled also. Additionally, the stack was made executable using the execstack tool. To make the attack-scenario more interesting, the owner was changed to root and the setuid flag was set on the executable files. Listing 6.3 illustrates this.
Listing 6.2: Compiling the Vulnerable Program

```
gcc -g -c -fno-stack-protector -o vuln.var2.o main.c
gcc -g -edummy_syscall -o vuln.var2 -T linker_script vuln.var2.o
gcc -g -fno-stack-protector -e move_stack -o vuln.var1 main.c
execstack -s ./vuln.var1
execstack -s ./vuln.var2
chown root vuln.var1
chown root vuln.var2
chmod u+s vuln.var1
chmod u+s vuln.var2
```

Listing 6.3: Setuid Bit is set for the Executables

```
marc@linuxmint ~/vuln ls -1 vuln.var
-rw-r-xr-x 1 root root 11690 Aug 7 14:14 vuln.var1
-rw-r-xr-x 1 root root 11690 Aug 7 14:14 vuln.var2
```

Diversification at link level (using the modified linker script in Listing 5.2) causes the segments of the executable to be mapped at different virtual memory addresses. Listing 6.4 shows a truncated dump of the executable’s headers. The first variant will have its memory segments mapped normally, because the default linker script was used during compilation. For the other variant, the modified linker script was used and so the segments are mapped in the upper partition of the process’ user-mode address space (commencing at address 0x68048000).

As shown in the listings 6.5 and 6.6, the mappings of the executable and of the stack are in their respective partitions. However, shared objects are still mapped to their default positions, and so do not comply with the partitioning. This issue remains to be solved, and is discussed in Chapter 5.3.

Listing 6.7 shows that the program operates normally when it is fed the “expected” input. When it is fed with malicious input, however, the vulnerability can be exploited to give an attacker root privileges. The attack vector passed as parameter contains a NOP-sled (77 bytes), shellcode (code that loads a shell into the process’ address space using the execve system call), and a return address pointing to the NOP-sled. The NOPs allow some inaccuracy in the fake return address, and have the additional purpose of padding it. The screenshot also shows that the same attack vector does not work for the other variant, as the memory at the return address is not mapped, because the stack is at its original position in the second variant (it starts at 0xc0000000). As can be seen in the attack vector, the return address used is 0x68046f7e. The attacker has thus created the attack vector for the first variant, and to additionally exploit the other variant, the attacker would need to modify the return address appropriately. When the programs are run normally, this would not be an issue for the attacker, but in our replicated and diversified execution environment, the monitor would synchronize all input (and output), and thus the attacker could not feed the vulnerable programs separate attack vectors.

Listing 6.8 shows that the exploit does not work when the monitor is running the
Listing 6.4: Objdump of Executables

```
marc@linuxmint ~/vuln objdump -h vuln.var1 | egrep '
  9 .rel.plt 00000050 080483c8 080483c8 000003c8 2 2
  11 .plt 00000b0 08048440 08048440 00000440 2 4
  12 .text 000005d4 080484f0 080484f0 000004f0 2 4
  14 .rodata 00000071 08048ad8 08048ad8 00000ad8 2 2
  21 .got 00000004 08049ffe 08049ffe 00000ffe 2 2
  22 .got.plt 0000034 0804a000 0804a000 00001000 2 2
  23 .data 0000000a 0804a34 0804a34 00001034 2 2
  24 .bss 0000000c 0804a03c 0804a03c 0000103c 2 2
```

Listing 6.5: Memory Mappings of First Variant at Runtime

```
marc@linuxmint ~/monitor cat /proc/4955/maps
08048000-08049000 r-xp 00000000 08:01 1469626
  /home/marc/vuln/vuln.var1
08049000-0804a000 r-xp 00000000 08:01 1469626
  /home/marc/vuln/vuln.var1
0804a000-0804b000 rwxp 00010000 08:01 1469626
  /home/marc/vuln/vuln.var1
68046000-68048000 rwxp 00000000 00:00 0
  [stack:4955]
68048000-94024000 ---p 00000000 00:00 0
b7ef0000-b7e10000 rwxp 00000000 00:00 0
b7e10000-b7fbd000 r-xp 00000000 08:01 1442630
  /lib/i386-linux-gnu/libc -2.17.so
b7fbd000-b7fb0000 r-xp 001ad000 08:01 1442630
  /lib/i386-linux-gnu/libc -2.17.so
b7fb0000-b7fc0000 rwxp 001af000 08:01 1442630
  /lib/i386-linux-gnu/libc -2.17.so
b7fc0000-b7fc3000 rwxp 00000000 00:00 0
b7fd0000-b7fdd000 rwxp 00000000 00:00 0
b7fdd0000-b7fde000 r-xp 00000000 00:00 0
  [vdso]
b7fde0000-b7ffe0000 r-xp 00000000 08:01 1442606
  /lib/i386-linux-gnu/ld -2.17.so
b7ffe0000-b7fff0000 r-xp 0001f000 08:01 1442606
  /lib/i386-linux-gnu/ld -2.17.so
b7ff0000-b8000000 rwxp 00200000 08:01 1442606
  /lib/i386-linux-gnu/ld -2.17.so
bfff0000-bffff0000 rwxp 00000000 00:00 0
```

51
Listing 6.6: Memory Mappings of Second Variant at Runtime

```
marc@linuxmint ~/monitor cat /proc/4973/maps
   /home/marc/vuln/vuln.var2
```

Listing 6.7: Exploit of one Program Variant

```
marc@linuxmint ~/monitor whoami
marc
marc@linuxmint ~/monitor ../../../vuln/vuln.varl MarcHello Marc
marc@linuxmint ~/monitor ../../../vuln/vuln.var2 Marc
Hello Marc
marc@linuxmint ~/monitor ../../../vuln/vuln.varl \( perl -e 'print "x90"'
```

Hello <unprintable characters>~/bin/sh<unprintable characters>

```
# whoami
root
# exit
```

```
marc@linuxmint ~/monitor ../../../vuln/vuln.var2 \( perl -e 'print "x90"'
```

Hello <unprintable characters>~/bin/sh<unprintable characters>

```
Segmentation fault
```
diversified variants concurrently. To see what is going on in the monitor, some debug output was enabled (using a preprocessor flag). Following the lines that print the process identifiers, the eax register is shown at the return of each system call. orig_eax shows the system call number and eax shows the return value. It can be seen that the program performs the printf operation on line 18 of the code in Listing 6.1. printf internally calls the write system call (system call number 4), which returns the number of bytes written. The exploit normally redirects the execution flow on return of the greetuser function; this is supposed to happen after the printf. But as can be seen from listing 6.8, the monitor detects the divergence (the second variant receives a SIGSEGV) and shuts down both variants before the shellcode can be executed (no execve system call is issued). This shows the potential of diversified and replicated execution that protects against a wide range of memory error exploits, even without any other countermeasures enabled.
Listing 6.8: Exploit Attempt of Monitored Variants

```
marc@linuxmint ~/monitor/monitor-project ./monitor
   ../../vuln/vuln.var1 ../../../vuln/vuln.var2 (perl -e 'print '"\x90"x74"'
   "\xe8\x18\xe54\xd1xc0\x88\x46\xd07\x89\x76\x08\x89\x46\x0c"
   \x80\xe8\xe3" "\xff\xff\xff\xff\xff\x62\x69\x6e\x2f\x73\x68"')
Info: pid of monitor: 5238
Info: Working directory changed to /home/marc/vuln
Info: Path to executable of variant 0: vuln.var1
Info: Path to executable of variant 1: vuln.var2
Info: pid of variant 0: 5239
Info: pid of variant 1: 5240

Variant with pid 5239 has registers:
orig_eax: 197
eax: 0

Variant with pid 5240 has registers:
orig_eax: 197
eax: 0

Variant with pid 5239 has registers:
orig_eax: 192
eax: -1208115200

Variant with pid 5240 has registers:
orig_eax: 192
eax: -1208115200
Hello <unprintable characters>/bin/sh<unprintable characters>
Hello <unprintable characters>/bin/sh<unprintable characters>

Variant with pid 5239 has registers:
orig_eax: 4
eax: 123

Variant with pid 5240 has registers:
orig_eax: 4
eax: 123
Alarm: Variants potentially under attack.
Variant 1 with pid 5240 received a segmentation fault SIGSEGV.

Info: Variant 0 with pid 5239 terminated by monitor
Info: Variant 1 with pid 5240 terminated by monitor
marc@linuxmint ~/monitor/monitor-project
```
Replicated execution necessarily produces some performance overhead, as the CPU needs to process each instruction twice. Multiple additional context switches are also necessary per system call under monitoring. Also, monitor-variant communication has been completely implemented with \texttt{ptrace}, although shared memory is known to be the faster communication method. This remains to be implemented.

Although the variants were not diversified for the benchmark, performance measurements should nonetheless be accurate. This is because address-space partitioning does not cause any significant overhead, as diversification is carried out mostly at link time. The only overhead caused by diversification is the shifting of the stack at the beginning of the execution. Once the address space has been set up, there should be no additional overhead. The main performance impact is caused by the monitor, which forces every variant to stop on every system call and to wait until all other variants have reached this call. This causes multiple context switches and delays. Also, the transfer of data from one variant to another for synchronization is very slow when \texttt{ptrace} is used, since only 4 bytes of data can be transferred per system call, and so many context switches are necessary to transfer larger amounts of data.

GNU tar was used as benchmark to measure the runtime overhead. Benchmarks were performed in a VMWare virtual machine with 2 GB of RAM and a single processor running Linux Mint 3.8.0-19 x86 as the operating system. The host was a 64-bit Windows 7 Enterprise SP 1 system running on a quad-core Intel Core i5 M 540 CPU with 8 GB of RAM. To reduce noise caused by other system tasks, measurements were taken 3 times per benchmark, with the two test cases (with and without monitor) interleaving. The benchmark involved generating a .tar archive from the glibc-2.17 source code. glibc-2.17 contains 14,496 items, totalling 110.1 MB in size. To measure the execution time, the \texttt{time} command was used. The sum of the times the process spent in user and system mode was used as the result. The two commands shown in Figure 7.1 were used for runtime measurement under normal conditions and under replicated execution, respectively.

Listing 7.1: Benchmark with GNU tar

\begin{verbatim}
\texttt{time ./tar -cf archive.tar glibc-2.17/}
\texttt{time ./monitor ./tar ./tar -cf archive.tar glibc-2.17/}
\end{verbatim}
Measurements revealed that the monitored execution is about 150x slower than the unmonitored execution. As already indicated, this is probably because of the inefficiency of `ptrace` in data transfers. As shown in Chapter 4.4.3, `read` is the system call executed most often, with more than 50,000 executions during a single run of `tar -cf`. This makes archiving with GNU `tar` a worst-case scenario for our system. Every `read` is executed by one variant only, and thus the result buffer needs to be copied to the other variant. Utilizing shared memory for this operation instead of `ptrace` would significantly increase efficiency. Salamat et al. [75] [77] showed that shared memory is more than 900 times faster than `ptrace` for a 128 KB buffer. Another reason for this high overhead is that context switches are much slower in a virtualized environment [84].
Conclusions

This report has introduced some of the most common ways of exploiting memory errors and the current state of countermeasures. Operating systems have made significant progress and it is now much harder and more time-consuming for an attacker to exploit memory errors. However, memory errors are still widely exploited, sometimes as a result of weaknesses in the OS’s protections, or else from the negligence of users or application developers. In many cases, security is traded against performance and usability, which leaves some countermeasures unused.

The research conducted during this work demonstrates both the potential and limitations of the diversified and replicated execution of programs for application security. The diversification techniques analysed all have advantages and disadvantages, and total security from memory-error exploitation is difficult to achieve. The potential of address-space partitioning is very great, and includes the deterministic prevention of attacks which perform full pointer overwrites. Such overwrites are conducted by the majority of memory-error exploits. It has been shown, however, that this approach cannot protect deterministically against partial pointer overwrites, even if address space shifting is applied: additional countermeasures are necessary to provide overall protection. Since replicated execution and monitoring produces overhead, the environment, its security requirements, and its resources must all be taken into account in deciding whether such a system is realistic for productive use.

The pros and cons of the concepts developed in previous research have been analysed, and some flaws in their approaches demonstrated. In the course of this research, a list of all Linux system calls, their parameters, return values, and the options for handling them with the monitor has been created. The implementation shows how diversification with memory partitioning and monitoring can be implemented. Although open issues remain, this work aims at a generic implementation of such an execution environment that can be used for an arbitrary number of variants and with any diversification method. It has been shown that address-space partitioning can be carried out using a modified linker script and a statically linked function. A modified linker script can relocate the segments that are present in the executable. Other segments, such as the stack, need to be relocated at runtime. The code for relocating the stack works well for statically linked binaries.

The proof-of-concept demonstrates that the ideas presented work in practice, and
that such an execution environment deserves further research. Such research could focus on the issues described in Chapters 4.5 and 5.3.

The main contribution of this work is its thorough comparison of the various studies conducted in this area, an analysis of the problems and potential solutions, and a generic implementation that demonstrates the concept and which can be used as basis for further research.
Bibliography


[83] Matt Conover (Shok) and the w00w00 Security Team. w00w00 on Heap Overflows. http://www.cgsecurity.org/exploit/heaptut.txt, January 1999.


A.1 System Call Statistics

The following scripts were used to collect information about system calls at runtime and to order the results:

Listing A.1: System call trace for Mozilla firefox

```
strace -f -o ~/strace_firefox.txt firefox
```

Listing A.2: System call trace for GNU tar

```
strace -f -o ~/strace_tar.txt tar -cf archive.tar
  "linux -2.6.16-openvz/
```

Listing A.3: System call trace for Apache/2.2.22

```
ab -n 1000 -c 10 -g ~/tmp.txt
  "localhost/GetSimpleCMS_3.2.1/index.php
sudo -i
/etc/init.d/apache2 restart
ps auxw | grep sbin/apache | awk '{print"-p \" 2}'' | xargs
  "strace -f -o '/home/marc/strace_httpd.txt'
```

Listing A.4: Script to group strace output by system call and rank by number of occurrence

```
# Counts the number of system calls in an strace output
  "file and ranks by number of occurrence of a system call
# First argument is the input file (strace output) and
  "second argument is the output file
sed 's:[^a-z]([(a-z0-9][a-z0-9_])\.:\:1:g' <1 |  sort |
  "uniq -c | sort -nr >2
```
A.2 The Project’s Code Repository

The code is distributed under the terms of the GNU Lesser General Public License (LGPL). It can be found online: http://www.mni.at/mscproject/