JITSec: Just-in-time Security for Code Injection Attacks

Willem De Groef, Nick Nikiforakis, Yves Younan, and Frank Piessens

IBBT-DistriNet
Katholieke Universiteit Leuven
Celestijnenlaan 200A B3001
Leuven, Belgium
willem.degroef@student.kuleuven.be
{nick.nikiforakis,yves.younan,frank.piessens}@cs.kuleuven.be

Abstract. In order to combat code injection attacks, modern operating systems use among others, the Write-XOR-Execute ($W\oplus$X) countermeasure which disallows code execution from writable memory pages. Unfortunately though, a widely used programming technique, namely Just-in-time compilation (JIT), clashes with the use of this countermeasure since its use relies on writable memory where it places dynamically generated code. In order to overcome this problem, programmers that use such techniques, code around the ($W\oplus$X) countermeasure in ways that open up their programs to code injection attacks.
In this paper we present JITSec, a new countermeasure specifically geared towards programs that use JIT compilation techniques. By decoupling sensitive from non-sensitive code, we block the execution of the former from writable memory pages while allowing the latter. Our distinction is based on the separation of normal function calls within the processes' address space from system calls. Our resulting system is fully transparent to the operating system and makes it possible to protect all applications without user intervention or access to source code. The overhead of our prototype is negligible (under 2%), allowing its adoption as an extra line of defense in cases where code execution from writable memory pages is desired.

Key words: code injection, execution monitor, just-in-time compilation

1 Introduction

In 1988, the most popular worm in the history of computer worms, the Morris worm, was unleashed to the Internet [22]. Its main spreading mechanism was the exploitation of a buffer overflow vulnerability present in the finger daemon in UNIX systems. Apart from the negative consequences resulting from its spreading, the worm attracted the world's attention to buffer overflows and to the potential resulting from their exploitation. In 1996, Elias Levy released a whitepaper explaining the specifics of stack-based buffer overflows[12], with the title “Smashing the stack for fun and profit” which quickly became one of the most read papers amongst the computer security community.

Buffer overflows belong to a broader range of exploitation techniques, collectively known as “code injection attacks”. Code injection attacks involve an attacker injecting machine code that was not present in the original application and diverting the execution of the vulnerable application into that code. Such attacks are possible in a number of compiled languages and their effects can range all the way from information leakage to complete access over a host running the vulnerable application. Last year has been a showcase for code injection attacks since many high-profile companies like Google, Yahoo and Symantec were attacked by zero-day code injection exploits for major software products such as Internet Explorer from Microsoft and Acrobat Reader from Adobe Systems [1, 14]. These vulnerabilities allowed attackers to remotely execute code on the vulnerable hosts.

While safe languages that are not vulnerable to code injection attacks have existed for years, unsafe languages such as C and C++, are still widely used mainly due to their increased performance and the existence of legacy code. application written in unsafe languages are generally much faster than their counterparts written in safe languages. Also, millions of lines of C code exist in application that need to be updated and maintained. At the time of writing, the TIOBE Programming Community Index, ranks C and C++ as the second and third most popular programming languages [24]. Together, they account for more than 28% of all computer languages, 10% more than the currently first-ranking, Java.

Due to the sustained popularity of the languages, many countermeasures have been proposed by academics over the last 15 years and a number of them have reached production level, by being by default included in modern operating systems. The three most popular and complementing countermeasures are (i) Address Space Layout Randomization (ASLR) [17], (ii) non-executable stack and heap [20] and (iii) probabilistic protection of stack frames (e.g. StackGuard [6]). Today, most modern operating systems ship with a combination of all or some of the above countermeasures active. The ASLR and StackGuard type countermeasures are considered fast and efficient however their protection relies on the secrecy of memory and thus they suffer from memory leakage attacks [23]. On the other hand, the non-executable stack and heap countermeasure is a non-probabilistic countermeasure that protects applications based on the policy that a memory page can either be writable or executable but not both (commonly abbreviated as $W\oplus$X). Thus when an application is vulnerable to a code injection attack and the attacker tries to divert the application’s execution flow to code that he injected earlier in writable memory pages, the processor will not execute that code and will force the process to terminate, effectively stopping the attack.

In this paper we investigate how the $W \oplus X$ countermeasure works and we point out the limitations that it places on legitimate programming techniques[1], such as Just-in-time (JIT) compiling. We argue that an application can maintain its security properties and execute code from the stack and heap by decoupling sensitive from non-sensitive code and allowing the latter to run from writable memory pages. We realize this idea with JITSec, a modification of the GNU/Linux kernel which transforms the system-call interface to be callsite-aware. Every time that an application interrupts to the kernel with the purpose of executing a system call, JITSec checks the callsite of the system call. If the system call originated from the original `.text` section of our process or from shared libraries it is allowed to continue, else our system forces the process to exit. This mechanism is based on the observation that, while code can be dynamically generated and executed from writable pages, a system call directly from the stack or from the heap is a sign that an attacker has successfully diverted the execution flow of the application and is now trying to leverage his control to gain more access, e.g. by requesting the execution of a shell by using the `execve()` system call. Our system imposes negligible time and memory overhead and is geared towards processes that want to be able to run code from writable memory pages but not at the expense of security. The combination of JITSec with ASLR and a StackGuard-type countermeasure provides more security than the current state of the art for processes that use JIT compilation techniques.

The rest of this paper is structured as follows: In Section 2 we briefly discuss all of the background information related to our countermeasure and we give an example of a code injection attack. In Section 3 we present the design of our JITSec countermeasure followed by an evaluation of it in Section 4. In Section 5 we discuss related work and we conclude in Section 6.

2 Background

JITSec is a security mechanism aimed at protecting processes which make use of Just-in-time compiling techniques but still want to be protected against code injection attacks. In this section, we will briefly discuss the concepts of JIT compilers and the system call mechanism in the GNU/Linux kernel. We will also give an example of a code injection attack and provide our model for an attacker.

2.1 JIT compilers

Overview. Two main categories that programming languages can fall into are: a) compiled and b) interpreted. When a program is written in a compiled language, the output program uses architecture-specific instructions that can run natively on a specific processor (such as `x86`). On the other hand, interpreted

[1] It is worth noting that in older GNU/Linux kernels, an executable stack was used for signal handling. In recent versions signal handling is achieved without the use of an executable stack.

programs are source code files that get interpreted and executed line-by-line at each run time of the program. Just-in-time (JIT) compilers are a hybrid between interpreted and compiled programs. The interpreter of an interpreted language can choose fragments of interpreted code from the program, convert them to executable code and use these executable code fragments instead of the interpreted code fragments. This transformation is mainly performed to decrease the execution time of frequently used parts of an interpreted program. For example, a JavaScript engine can observe that a a sequence of instructions is called multiple times throughout the script (known as a hotpath) and choose to transform it into executable code [8]. All of the subsequent invocations of that hotpath will cause the execution of native code running directly on the processor instead of interpreting it over and over again.

Security Implications. While JIT compilers can improve the execution time of interpreted programs, their functioning inherently clashes with one of the most used attack mitigation techniques, namely $W \oplus$X [21]. When an interpreter creates executable code, it must be stored in memory so that it can be used by later parts of the program. However $W \oplus$X states that a memory page can either be writable or executable but not both. Thus, the executable code that was deliberately generated by the JIT compiler will be unable to run as long as $W \oplus$X is actively protecting a process. Programmers circumvent this problem by either de-activating the countermeasure completely for a specific process or by marking a number of memory pages explicitly as executable through the appropriate system call.

In both cases, an attacker can use this behavior to convince the JIT compiler to place his code in executable memory so that when he later on diverts the execution flow of the program to them, the processor will execute the injected code instead of terminating the process.

A real exploitation of this attack scenario was presented at the BlackHat DC 2010 conference against a fully-patched Microsoft Windows 7 host. The attack uses the JIT compiler engine of ActionScript (scripting language developed by Adobe Systems) to place code in executable pages despite the DEP[2] countermeasure being active on the machine [3].

2.2 System Call Mechanism

System calls are the interface between user programs and the operating system. Any functionality that a program needs to do, which is outside its process scope (e.g. write a file, send a packet over the network, communicate with other processes etc.) is done using a system call. The main reason for this separation is for the operating system to enforce security policies and conform processes to their permissions. In the `x86` architecture and more specifically in the GNU/Linux OS, the system call interface is a specific and well defined

[2] DEP (Data Execution Prevention) is the equivalent Microsoft Windows protection of $W \oplus$X

sequence of events. When a program wants to communicate with the kernel through a system call, it places the number of the system call in the `eax` register and the arguments for the system call in registers `ebx,ecx,edx,edi` and `esi`. If the system call has more than six arguments (including the system call number) then the rest are passed to the kernel through the process's stack[3]. Once the arguments are placed in the registers, the program issues a system call interrupt (`int 0x80`) which informs the operating system that the caller program wants to perform a system call. The kernel uses the system call number stored in the `eax` register to dispatch the correct system call. Once the system call is finished, the results are populated to the caller (using the return code stored in `eax` and possibly additional memory in the address space of the caller process) and the execution flow is returned.

2.3 Heap Overflow

Code Listing 1 Code snippet vulnerable to a heap overflow

```
struct data_node {
    char src[128];
    char dst[128];
    int (*transform_func)(char *, char*);
};

int main (int argc, char *argv[]) {
    struct data_node *n;
    int i;
    ...

    n = malloc(sizeof(struct data_node));
    n->transform_func = capitalize;

    for(i=0; argv[1][i] != '\0'; i++)
        n->src[i] = argv[1][i];

    (*n->transform_func)(n->src, n->dst);
    ...
}
```

Code injection attacks are attacks where an attacker can inject machine code into the address space of a process and then manage to divert the execution flow

[3] Most times, the programmer will not have to explicitly move values to the registers e.g. the standard shared C library `libc` contains wrappers for all system calls so that system calls can be called as normal functions.

of the program to his code. In compiled languages such as C and C++, code injection attacks range from standard stack- and heap-based buffer overflows to dangling pointers and return-to-libc attacks. Code Listing 1 is vulnerable to a code injection attack. The purpose of the program is to read a string from the user, perform a transformation on that string and then save it along with the original string. Since there could be many transformations, the transformation function is called through a function pointer which is set by the programmer before the copying of the string. The code that reads the string from the execution environment is vulnerable to a heap overflow since it doesn't perform any checks whether the `src` buffer is large enough to hold the contents of the command line argument. If the attacker provides a string that is longer than 128 bytes, the string will spill out to the `dst` buffer. If the provided string is longer than 256 bytes then the string will also overwrite the function pointer that is called in the next line. The attacker can simply enter his shellcode in the `src` buffer and overwrite the function pointer with the address of the buffer. Thus the program, instead of calling `capitalize`, will call attacker-provided shellcode which can lead to a complete compromise of the host.

2.4 Attacker Model

In this work we assume that a vulnerability exists in a running program that will allow a local or remote attacker to change at least one memory location in the userspace of the process which will be later on used by the program to transfer the control-flow (e.g. a heap overflow, see Sec. 2.3). Examples of such memory locations are return addresses, function pointers, entries in virtual function tables and so on. We also assume that at least one attacker-controlled variable exists which the attacker can use to inject the code of his choice. The variable has to be large enough to fit the attacker's code and it has to be located on the stack or on the heap of the program. These two assumptions together, allow an attacker to first place code of his choice in a variable and then later on transfer the control flow of the program to that code. Using his newly-found control of the program's execution flow, the attacker will try to gain more control over the vulnerable host by issuing system calls which will execute with the privileges of the running process. In our model, the attacker issues these system calls from his previously injected code which is located on the stack or the heap of the running process.

3 Design & Implementation

This section presents a formal description of our countermeasure, JITSec, followed by implementation details of our prototype. The prototype was implemented as a kernel module on an Ubuntu GNU/Linux distribution with kernel version 2.6.31. The prototype source code is publicly available at [9].

The general idea behind the approach is to find a way to determine the callsite of a specific system call. Based on the specifics of the `x86` architecture, the first subsection describes how one can find this callsite, via a detour, by using the value of the saved `eip` register of the calling process. The next subsection depicts how these findings can be transformed into a monitor.

3.1 Formal Description

A process p has a virtual address space Θ_p that contains a certain number of non-overlapping memory regions μ_j called segments. A process can use a finite group of memory registers γ_i : register γ_{eip} contains the address of the next-to-execute instruction and register γ_{eax} holds the system call number.

A certain system call interrupt `int` has exactly one callsite κ_i (the subscript i is the system call number in γ_{eax}) in exactly one specific memory region μ_j.

When the interrupt instruction is being handled by the kernel, no control flow changes can be made by the generating process (the `int` instruction doesn't change γ_{eip}). Since γ_{eip} points to the next instruction after the interrupt instruction, the callsite κ_i and the address in γ_{eip} are in the same memory region. This leads to the observation that one can check the callsite of a specific system call interrupt by examining the contents of γ_{eip}.

3.2 Monitor

The monitor needs to check the callsite of a system call interrupt in order to guarantee correct behavior of the interrupting process. When the callsite κ_i of a specific system call interrupt lies on the stack or the heap, the request is modified to terminate the application (i.e. i gets changed to *exit* by replacing the value in γ_{eax} with the value of the exit system call). In all other cases, the system call will not be changed.

This leads to the formal description of the monitor

$$Monitor(i) = \begin{cases} \gamma_{eip} \in \mu_{stack} & \longrightarrow exit \\ \gamma_{eip} \in \mu_{heap} & \longrightarrow exit \\ otherwise & \longrightarrow i \end{cases}$$

with the new value $\gamma_{eax}^{'} = Monitor(\gamma_{eax})$.

3.3 Kernel Module Details

Once the kernel module for JITSec is installed, it places itself between the user processes and the system call handler by hijacking the `IDT` register[4] and overwriting the system call handling entry with the address of a trampoline function (see Code Listing 2). The trampoline function is a small function written in assembly code whose purpose is to store the contents of the registers on

[4] The Interrupt Descriptor Table (IDT) is a data structure used to implement the interrupt vector table. The `IDT` register is used to determine the correct function to handle interrupts and exceptions.

Code Listing 2 JITSec trampoline function

```
void asmlinkage jitsec_trampoline (void) {
    __asm__ __volatile__ (
            "push %edi              \n"
            "push %esi              \n"
            "push %edx              \n"
            "push %ecx              \n"
            "push %ebx              \n"
            "push %eax              \n
            "call jitsec_monitor    \n"
            //discard original eax value
            "pop %ebx               \n"
            "pop %ebx               \n"
            "pop %ecx               \n"
            "pop %edx               \n"
            "pop %esi               \n"
            "pop %edi               \n"
            "pushl orig_sc_routine   \n");
}
```

the stack, call the actual policy-enforcing monitor function (`jitsec_monitor`), restore the contents of the registers and call the original system call handling function. The contents of registers `eax, edi, esi, edx, ecx` and `ebx` are saved and restored in order to undo any values that could have been replaced by our policy-enforcing function. The results of the `jitsec_monitor` are propagated and enforced through the use of the `eax` register and thus this register is not restored.

The policy-enforcing function (see Code Listing 3) uses the appropriate data structures provided by the GNU/Linux kernel in order to access the return address where the control flow is supposed to return to when the system call finishes. Once this address is located, it is checked against the range of the stack memory section and the heap memory section of the calling process. If the address is within one of the two ranges, the function substitutes the contents of the `eax` register, which is the number of the original system call as requested by the running process, with the number for the `exit()` system call. On the other hand, if the return address does not fall in any of the two ranges, the original `eax` value is kept. Once the policy-enforcing function returns and the original system call handling function is called, the contents of the `eax` register will be used in order to dispatch the execution control flow to the appropriate system call. If the policy-enforcing function of JITSec decided that the system call was a result of a code injection attack, the `eax` register will contain the number for the `exit()` system call and thus the kernel will cause the process to exit instead of executing the attacker-requested system call.

Code Listing 3 JITSec policy-enforcing function

```
#define K_EXIT  0x1 // exit system call number
#define IS_BETWEEN_ADDR(start, log2size, addr)
      (((start ^ addr) >> log2size) == 0)

int jitsec_monitor (unsigned int orig_eax)
{
    /* general structures provided by the kernel */
    struct task_struct *task = current_thread_info()->task;
 struct mm_struct *mm = task->mm;
 struct pt_regs *r;

 /* contains the (possibly) new eax value */
 unsigned int new_eax = orig_eax;

 /* Size of stack is given in memory pages
    log_2(stack_size * page_size) =
    log_2(stack_size) + (log_2(4096) = 12) */
 unsigned long l = log_2(mm->stack_vm) + 12;

 /* accessing registers for the current process thread */
 r = ((struct pt_regs *) task->thread.sp0 - 1);

 /* 1. Check if callsite on stack */
 if (IS_BETWEEN_ADDR(mm->start_stack, l, r->ip))
 {
     /* callsite on stack -> exit! */
     new_eax = K_EXIT;
 }
 else
 {
     /* subtracting start and end gives size of heap */
  l = log_2(mm->brk - mm->start_brk);

  /* 2. Check if callsite on heap */
  if (IS_BETWEEN_ADDR(mm->start_brk, l, r->ip))
  {
      /* callsite on heap -> exit! */
      new_eax = K_EXIT;
  }
 }

 return new_eax;
}
```

4 Evaluation

This section reports on the performance of the presented countermeasure and gives an evaluation of its security properties. The benchmarks were executed on a machine with an Intel®Core^{TM}2 Quad CPU @ 2.40GHz with 4096 MB RAM (swapping disabled) running Ubuntu 9.10 with kernel version 2.6.31.

4.1 Performance Overhead

Time Overhead. The first part of the evaluation is about time overhead.

Table 1. Micro benchmarks (Average number of system calls performed in 2 seconds)

System call	Without JITSec	With JITSec	**Overhead**
`getpid`	7791687	6879281	**11.71%**
`clone`	4374	4363	**0.25%**

The micro benchmarks measure the impact of JITSec on individual system calls (see Table 1). These results were collected by a repeated execution of every system call within a time window of 2 seconds. Both the performance of simple system calls (e.g. `getpid`) and complex system calls (e.g. `clone`) were measured. As expected, the overhead of 11.71% for simple system calls is significantly higher than the 0.25% overhead for complex system calls (where the high costs of buffering and memory accesses mask the overhead introduced by JITSec). The overhead of simple system calls can be considered the upper-bound of JITSec's imposed overhead over normal programs.

Table 2. Macro benchmarks (SPEC CPU2000 Integer)

Program	Without JITSec	With JITSec	**Overhead**
164.gzip	195	204	**4.41%**
175.vpr	148	148	**1.33%**
176.gcc	91.50	96.10	**4.79%**
181.mcf	94.27	94.57	**0.32%**
186.crafty	105.33	105.33	**0%**
197.parser	1332	1341	**0.67%**
253.perlbmk	133.33	133.33	**0%**
254.gap	125	127.33	**1.83%**
256.bzip2	214.33	224	**4.32%**
300.twolf	258	260	**0.77%**
Average overhead			**1.84%**

To measure the general performance overhead of JITSec on applications, we've used the SPEC CPU2000 Integer benchmark suite [10] (see Table 2 for the average execution time in seconds for all tests within the suite). The macro benchmarks show that the average overhead of JITSec is less than 2% for real-world programs. Since JITSec operates in the kernel-level and is invoked as a result of a system call, programs that perform more arithmetic operations and less system calls will have less overhead than "system call-dense" programs. In total, the overhead of JITSec is below the overhead of most other system call monitoring systems.

Memory Overhead. Memory overhead is determined by (i) the possibly larger memory footprint of applications (in memory and on disk) and (ii) the introduction of the countermeasure itself. The memory space on disk does not change after using JITSec (unlike other countermeasures that change the binary of an application) nor does the memory footprint of protected applications. The actual JITSec monitor occupies about 180 bytes of kernel memory. Every system call temporarily requires some extra stack space (because of the space needs in the `jitsec_trampoline` function) which is immediately reclaimed by the process once the function returns.

4.2 Security Properties

The presented countermeasure offers no protection against the process of injection itself, though it protects effectively against execution of system calls originating from the injected code. In comparison with many other countermeasures, JITSec (i) doesn't require changes to applications or libraries (source code is not required) and (ii) is compatible with other countermeasures. Because of the implementation via a kernel module, the prototype can easily be integrated on running and/or existing systems. These properties allow for a low setup cost and enable an easy transition from a non-protected system to a protected one.

The security of the presented countermeasure can be summarized in the next three security properties:

JITSec protects all applications. Since JITSec works on kernel level, all applications (running and yet-to-run applications) will be automatically protected immediately after deployment of the countermeasure.

JITSec is non-bypassable. Since the `x86` architecture clearly defines that interrupts and system call interrupts are not possible in any other way and since the `eip` register can not be edited directly, JITSec can be considered non-bypassable.

JITSec protects against attacks of the attack model. Due to the fact that no learning phase is required, that the callsite of a system call can be calculated in a straightforward way and together with the properties stated above, one can conclude that JITSec offers protection against system calls originated from the stack or heap.

The presented countermeasure does not offer protection against the following attacks:

- Non-control-data attacks. This kind of attacks don't use system calls directly but focus on changing application data instead of process management data [4].
- Return-into-libc attacks. These attacks use existing program code and libraries instead of injecting new code in a process' address space. JITSec can not protect against such attacks because system calls originate from legitimate memory segments.
- Specific kernel injection attacks. If there is a vulnerability in the kernel which allows a potential attacker to make direct changes to kernel memory then the correct behavior of JITSec can not be guaranteed anymore.

The current exploitation methodologies show that attackers prefer to use the return-into-libc method of attacking only as a first step towards the total exploitation of a vulnerable program. This is because, creating a complete and custom payload out of return-into-libc calls is a hard and non-reliable process [15]. Thus the current state of the art in exploiting memory corruption vulnerabilities involves an attacker performing a return-to-libc attack to mark a memory page as executable, place his shellcode on that page and then divert the execution flow to that page [2]. While JITSec can't protect against the return-to-libc attack itself, it will stop the actual attack since the injected code will be on the address range of the stack or heap.

5 Related Work

Many countermeasures exist that protect systems from code injection attacks. In this section we will focus on the countermeasures that are most closely related to our approach: countermeasures that will prevent or detect the misuse of system calls in an application. A broader survey of related work can be found in [27, 7, 26].

Execution monitors Monitoring of system calls falls within the broader category of execution monitors. Several types of execution monitors exist: ones that will enforce a specific policy on an application, preventing it from doing anything that is not allowed by the policy, while others will detect anomalies in the runtime of the application, e.g. is the application doing a specific system call while in a normal run of the application other system calls would be executed first . Both types of execution monitors can suffer from mimicry attacks[25]: if the attacker can mimic the behavior of the original program he can bypass the monitor. More granular monitors could be built, which in the extreme could prevent the attacker from executing anything but the original intended code of the program in the correct order. However, such extended monitoring could lead to higher false positives and/or higher performance overheads. JITSEC does not suffer from such mimicry attacks, nor does it suffer from false postivies, as the policy does not rely on any learned behavior of the program.

Kc et. al [11] propose a monitor similar to JITSec . They also provide wrappers over `libc` function calls in order to protect against return-to-libc attacks. The problem however is that the techniques used by the wrappers to thwart mimicry attacks can be circumvented by a determined attacker who has control of the execution flow. An attacker who manages to execute a single system-call (specifically a `mmap`) can change the permissions of the memory page containing his shellcode to "only-readable" and thus confuse their kernel-level monitor. JITSec on the other hand doesn't rely on permissions of pages but rather on address ranges which makes the attacker's `mmap` call ineffective.

Rabek et al. [18] perform static analysis on executables and will execute the location at which calls to Win32 API's are made. At runtime it will monitor if the location of the code calling the API is correct. However, the technique relies on the correctness of the return address on the stack. If the attacker places the expected return address on the stack, the countermeasure could be bypassed. Additionally the countermeasure relies on a user-mode wrapper for API calls, hence it could be bypassed by calling the function directly or by executing system calls directly from the injected code. The static analysis may also introduce false positives.

The technique discussed by Linn et al. [13] adds semantical information about the locations of legitimate system call instructions to the binary of the application. It requires the binary to be statically linked and it has to be able to analyze the application using static analysis, which may not always be the case.

Probabilistic approaches Countermeasures that rely on some kind of randomness to prevent the misuse of system calls can be termed probabilistic countermeasures. However the problem with these approaches is that they also rely on memory secrecy, which can not always be guaranteed[23]. If an attacker is able to print out memory locations, through an information leak (e.g. a buffer overread or a format string vulnerability) than an attacker may be able to bypass these types of countermeasures.

The technique suggested by Chew and Song [5] introduces randomness in the implementation of system software. One of the methods suggested changes the mapping between system call numbers and system call handler functions by mixing the system call table based on random numbers. This approach requires both changes to the kernel and binary rewriting of the application. A drawback of this approach is that only one mapping can be used for each application and this mapping remains unchanged, except for explicit recompilation of the kernel. Another disadvantage is that third party software can not be protected.

Oyama and Yonezawa[16] use cryptography for protection: both the system call number and the arguments are encrypted in user space and decrypted in the kernel. To achieve this, changes must be made to both the standard C library and the kernel. This approach has several disadvantages: (i) it can only protect dynamically linked applications, (ii) it is not compatible with other system call interceptors and (iii) determined attackers may also be able to find the random seed used for encryption in the program memory.

Rajagopalan et al. [19] also use cryptography. System calls get extra arguments that describe the policy of the system call and a message authentication code (MAC) that is used to verify the integrity of the policy and system call arguments. An installation program reads the binary of the application, generates policies and rewrites the binary to implement authenticated system calls. Disadvantages of this approach are that it requires relocatable statically linked binaries and that protected applications may be vulnerable to advanced mimicry attacks.

6 Conclusion

For more than 20 years, attackers have been abusing software vulnerabilities to convince hosts to execute malicious code of their choosing. Despite the adoption of several countermeasures in modern operating systems, certain programming techniques force applications to operate in a non-secure way, circumventing the protections offered by the operating systems. In this paper we presented the issues between the $W \oplus$X countermeasure and the Just-in-time compilation techniques and showed how a programmer is currently forced to choose between less security (disabling the $W \oplus$X countermeasure) or less functionality (not using JIT techniques). We argued that there is a way to allow execution from the stack and heap while preserving the security of the running application and we discussed the notion of decoupling sensitive from non-sensitive code. We realized our idea by designing and implementing JITSec, a countermeasure that allows the execution of non-sensitive code from writable memory pages but blocks the execution of system calls (sensitive code) from such pages. Our resulting system provides enhanced protection for processes that use JIT techniques with less than 2% of overhead. JITSec adds an extra line of defense against code injection attacks making it an ideal replacement of the $W \oplus$X countermeasure when code execution from writable memory pages is desired.

Acknowledgements

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

1. Adobe. Security bulletins and advisories.
2. Alexander Sotirov. Modern Exploitation and Memory Protection Bypasses. Invited Talk on USENIX Security, 2009.
3. Dionysus Blazakis. Interpreter Exploitation: Pointer Inference and JIT Spraying. In *BlackHat DC*, 2010.
4. S. Chen, J. Xu, E. C Sezer, P. Gauriar, and R. K Iyer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*, 2005.

5. M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical report, Carnegie Mellon University, December 2002.
6. Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, 1998.
7. Úlfar Erlingsson, Yves Younan, and Frank Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer, 2010.
8. Andreas Gal et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM conference on Programming language design and implementation*, 2009.
9. Willem De Groef. JITSec: Just-in-time Security for Code Injection Attacks - prototype source code. `http://www.cqrit.be/jitsec/`, 2010.
10. J. L. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, July 2000.
11. Gaurav S. Kc and Angelos D. Keromytis. e-NeXSh: Achieving an effectively non-executable stack and heap via system-call policing. In *Annual Computer Security Applictions Conference*, 2005.
12. Elias Levy. Smashing the stack for fun and profit. *Phrack magazine*, 7:49, 1996.
13. C. M Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K Debray, and J. H Hartman. Protecting against unexpected system calls. In *14th on USENIX Security Symposium*, 2005.
14. Microsoft. Security advisories.
15. MITRE. CVE-2010-2883. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2883, September 2010.
16. Y. Oyama and A. Yonezawa. Prevention of code-injection attacks by encrypting system call arguments. Technical report, University of Tokyo, March 2006.
17. PaX. Documentation for the PaX project.
18. Jesse C. Rabek, Roger I. Khazan, Scott M. Lewandowski, and Robert K. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. In *ACM workshop on Rapid Malcode*, 2003.
19. Mohan Rajagopalan, Matti Hiltunen, Trevor Jim, and Richard Schlichting. System call monitoring using authenticated system calls. *IEEE Transactions on Dependable and Secure Computing*, 3(3):216–229, June 2006.
20. Solar Designer. Non-executable user stack.
21. Alexander Sotirov and Mark Dowd. Bypassing Browser Memory Protections. In *Proceedings of BlackHat 2008*, 2008.
22. Eugene H. Spafford and Eugene H. Spafford. The internet worm program: An analysis. *Computer Communication Review*, 19, 1988.
23. Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security*, 2009.
24. TIOBE. Programming community index, 2010.
25. David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *9th Conference on Computer and Communications Security*, 2002.
26. Yves Younan. *Efficient Countermeasures for Software Vulnerabilities due to Memory Management Errors*. PhD thesis, Katholieke Universiteit Leuven, 2008.
27. Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical report, Katholieke Universiteit Leuven, July 2004.