Strengthening diversification defenses by means of a non-readable code segment

Sebastian Österlund
Department of Computer Science
Vrije Universiteit, Amsterdam, Netherlands
Supervised By: H. Bos & C. Giuffrida

***Abstract*—In this paper we present a new defense against Just-In-Time return-oriented-programming attacks. By making program code non-readable, the assembly of Just-In-Time gadgets by scanning the memory is effectively blocked. Using segmentation on Intel x86 hardware, the implementation of execute-only code can be achieved. We discuss two different ways of implementing such a defense for 32-bit Intel architecture: one for position dependent executables, and one for position independent executables. The first implementation works by splitting the address-space into two mirrored segments. The second implementation creates an execute-only memory-section at the top of the address-space, making it possible to still use the whole address-space. By relying on hardware segmentation the run-time performance overhead of these defenses is minimal.**

Keywords*—ROP, segmentation, XnR, buffer overflow, memory disclosure.

I. Introduction

In recent years traditional remote code execution exploits, making use of buffer-overflows, have become obsolete due to the introduction of data *execution prevention* (DEP) techniques. Techniques such as Intel NX [1] enforce a policy of making writable data non-executable. On the other hand, attacks such as *return-to-libc* and other Return-Oriented-Programming attacks overcome such techniques by executing (snippets of) existing code which necessarily has to be executable [2], creating a so-called *gadget* chain. An overview of how these attacks work will be given in Sec. II.

A common defense against ROP-exploits is Address-space layout randomization (ASLR). ASLR mitigates many of these attacks by randomizing the layout of a program loaded in memory. However, there are several known weaknesses in the currently used ASLR implementations [3].

For example a leaked pointer disclosure may give away the base-address of executable code (e.g. the base address of libc), thus making it possible to simply add an offset in the pre-assembled *gadget chain*. Fine-grained ASLR makes this harder [4][5], since the code is split up into smaller blocks and shuffled around. It has however been shown in [6] that, thanks to memory disclosure vulnerabilities, these gadgets can be assembled for a specific target by scanning the memory for code snippets. By creating such a *just-in-time* assembled exploit, the defense provided by *ASLR* is circumvented.

The contribution made by this paper is a segmentation-based defense against memory disclosure attacks. By relying on hardware segmentation, this approach should theoretically have no run-time performance overhead for Intel x86 architecture, once it has been set up. Both *position dependent executables* and *position independent executables* are covered. Furthermore an approach to implement a similar defense on x86_64 using the new MPX [7] instructions is presented. The defense mechanism we present in this paper is of interest, mainly, for use in security of network connected applications (such as servers or web-browsers), since these applications are often the main targets of remote code execution exploits.

II. Return-oriented-programming attacks

Remote code execution by means of buffer overflows has been a problem for a long time. In this section we give an overview of such remote code execution exploits.

By overwriting a function return address, code at an arbitrary location can be executed, thereby making remote code execution possible. The following code demonstrates a common exploitable bug in programs:

```
void vulnerable_function(char *user_input)
{
    char buffer[12];
    strcpy(buffer, user_input);
}

void hacked()
{
    printf("I have been hacked!!!!");
}

int main(int argc, char* argv[])
{
    vulnerable_function(argv[1]);
    return 0;
}
```

In the example above, the length of the string *user_input* is unknown. Since *strcpy* copies characters until it reaches a end-of-string character (i.e. NULL) [1], the program may overwrite memory beyond the *buffer*.

Using this buffer overflow, an attacker may overwrite the return address of the function *vulnerable_function*. By pointing

[1]Some compilers actually do object-size checking on strcpy, preventing some common buffer-overflows.

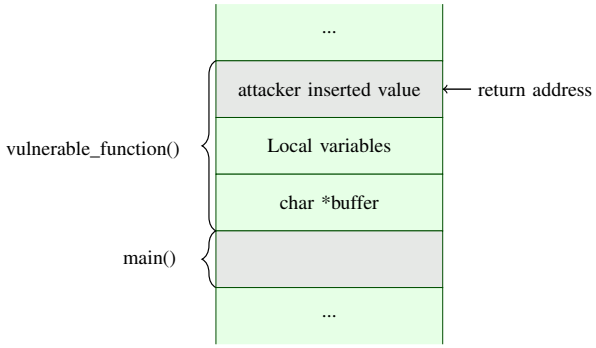


Fig. 1: **Example of buffer overflow attack.** strcpy overwrites the return address by an address entered by the attacker, executing code at a location chosen by the attacker.

this address to another function the execution of the program may be altered (See Fig. 2). Furthermore, an attacker may put arbitrary code into memory and point the return address to the entered code.

Luckily, *data execution prevention* techniques such as $W \oplus X$[2], making executable data non-writable, prohibit the execution of malicious code entered in this manner. When the processor tries to execute data on a page that is marked NX (non-executable) the processor throws an exception, terminating the offending program.

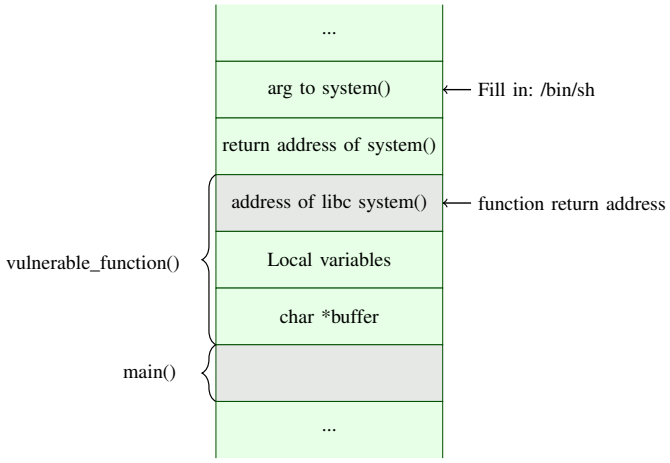


Fig. 2: **Example return-to-libc attack.** If the process runs as root, an attacker may gain access to a shell with root privileges. When the command system("/bin/sh") is executed the attacker has gained root access to the system.

Despite this protection it is still possible for an attacker to execute *existing* code. Since program code necessarily needs to be executable (why else call it a program...), it is possible to execute existing functions by placing its address (or any point in a procedure) in the return address. Usually this would not be critical, since the existing code in a program is limited in scope. However, an attacker may also choose to execute code from a library, giving the attacker much more available executable code to work with.

It is possible to chain together library functions by putting the address of the next function in the return address of the called function. By assembling such a chain, a so-called *gadget chain* is created. Furthermore, it has been shown in [2] that the GNU C library is *Turing complete*, thus any arbitrary program can be constructed by chaining together functions from *libc*.

III. Non-readable code

When gadget chains are assembled, the memory is scanned for pieces of executable code that can be "glued" together. For example an attacker may scan the memory for a *POP*-instruction followed by a *return*. By scanning the memory for such gadgets, a very targeted attack can be assembled *Just-in-time*, without knowing the exact location of executable code. If it, however, was prohibited to scan the memory, these attacks would be rendered useless. By enforcing a policy of making executable code non-readable (the abbreviation XnR will henceforth be used for this primitive) it becomes impossible to scan the memory for such gadgets. XnR together with ASLR, would make it very hard (or rather impossible) to determine where certain snippets of code are located, thus creating a strong defense against *JIT-ROP* attacks.

Sadly, implementing such a policy is not achieved by just marking executable pages non-readable. For Intel x86, memory that is executable implies that it is readable, thus XnR has to be implemented in another way. The obvious approach would be to emulate execute-only functionality on a per-page basis in software. Such an implementation is presented in [8] and will be discussed further in Sec. IX. In short: the problem with this approach is that software emulation adds some overhead, which is not necessarily required. In the next section a novel approach of achieving non-readable text without software-based page access emulation is presented.

IV. Segmentation-based access policy

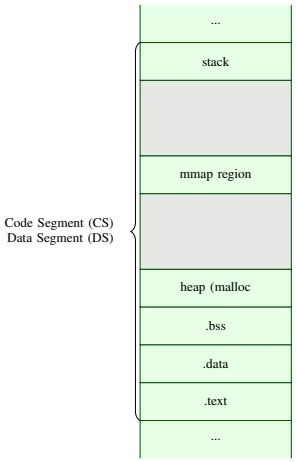
A. Segmentation on Intel x86

For the 8086 processor, Intel introduced segmentation. This memory-translation feature allows memory to be split into separate disjoint logical entities called segments. Segmentation allows programmers to create programs consisting of multiple segments located at different addresses in physical memory, without having to worry about at which address in physical memory certain parts of a program is loaded. The historical idea behind segmentation was to allow programmers to split the program into logically structured separated parts. There are, thus, two kinds of segment types: *data* and *code*. Data segments can only be used for reading/ writing memory, while code segments allow the memory-contents to be executed. For a more comprehensive description of segmentation see chapter 3.7 of [9].

When a valid segment descriptor has been loaded into a segment register, all access to memory is relative to the base

[2]Introduced by OpenBSD in 2003 (http://marc.info/?l=openbsd-miscm=105056000801065)

Fig. 3: **The layout of a standard ELF program in Linux** with a flat address-space. The Code Segment (CS) and Data Segment(DS) both cover the same area, namely 0x00000000 - 0xffffffff



address of the segment. If a memory address larger than the offset between the segment base and limit is loaded, the processor throws a general protection fault. Depending on which instruction is executed, the segment used for memory-translation differs. There are two different kinds of memory access patterns on Intel x86:

1) **Instruction fetch.** When a program instruction is fetched for execution by the processor, e.g. by means of a JMP. The *CS* (Code Selector) register is used to determine in which segment to access the address.
2) **Data load.** When a memory address is loaded through an instruction such as MOV. For these accesses the *DS* (Data Selector) register is used to determine in which segment to access the address. Normally, when using a flat memory model, it is also valid to load executable code using these instructions.

B. Process layout

In Linux, the virtual address space of a process is split up in different memory areas. Within each memory area, the access policy for the pages is the same. For example the data of a program is is put in one (writable) memory area, while the code is loaded into an executable memory area. For a standard ELF binary on Linux the layout of a loaded process looks approximately as in Fig 3. On Linux segmentation is not used; the kernel sets up two user segments USER_CS and USER_DS both starting at address 0x0 with a limit at 0xffffffff. They memory range covered by these two segments is, thus, equal to each other. This sets up a so-called flat memory model, in which code and data can be addressed interchangeably. The flat memory model is the preferred memory layout for most modern operating systems. Incidentally popular compilers for Linux, such as *gcc* and *clang*, assume a flat memory model.

C. Loading text in a separate segment

The main idea behind protection we propose is quite novel: load text into an address outside the limits of the user data segment, but inside a new code segment. When a read is performed on an address outside the user data segment, the processor will throw a protection fault, while allowing an instruction fetch from that same address.

A problem with splitting the memory in two segments is that compilers and linkers for Linux assume a flat memory model. When the Kernel loads the code, it assumes one contiguous piece of memory, where the access policy is determined by the page handler. Thus care should be given to load the correct section at the right memory address. Also, it would be desirable to not have to tinker too much with the layout of the programs. Ideally, one would want an existing binary should run with the proposed protection without having to be recompiled.

A very intuitive solution is to create two overlapping segments with starting at address 0x0, where the DATA-segment covers a subset of the CODE-segment but its limit is lower than that of the CODE-segment. This creates a section at the top of the address space that is execute-only. When using this approach, all executable code that should be non-readable is loaded in that particular section.

This approach was the initial solution tried while developing the proof-of-concept implementation. There is, however, a problem with this approach, namely *shared libraries*. Shared libraries are loaded into a contiguous piece of memory starting at an address chosen by the dynamic linker. This would not be a problem, since they can be forced to be loaded in the execute-only section, however, the static data section of the shared libraries are loaded into that same piece of memory. These data sections need to be readable for the libraries to work. While moving these data sections is possible, it is very cumbersome; all references to these memory addresses have to be updated by means of instrumentation. Also, relocating code in Position-Dependent-Executables is not trivial.

Another solution is to split the available virtual address space into two disjoint segments that mirror each other, with the exception that execute-only sections are nulled / freed in the DATA segment. Instruction fetches would be loaded form the CODE-segment, while memory loads would be loaded form the DATA-segment, giving the illusion of a flat address space. The obvious downside is that the available address space is cut in half. The upside is that a very generic defense mechanism can be implemented; when marking a sections as executable simply load it into the CODE-segment. This makes it possible to use existing memory protection infrastructure, such as *mprotect*.

With such an approach one could simply say that areas with the executable flag set are loaded into the CODE-segment and areas marked with the read/write flag are loaded into the DATA-segment. If an area is both executable and readable (and/or writable), it is loaded into both segments. By simply marking executable code sections in the binary as execute-only, the proper protection would be guaranteed.

To achieve such an implementation there are thus two tasks at hand:

1) Implement Execute-only memory primitive on Intel x86.
2) Mark executable code sections non-readable, so that the execute-only primitive is used.

D. Allowing legitimate text reads

Something that has to be taken into consideration when implementing the proposed protections, is that there are some legitimate reasons to read executable code. For example in the special case when the stack is made executable, the proposed defense should not take effect of the stack. Also there are some headers located in the text section of shared libraries that need to be readable [8]. The problem can be solved by recompiling the libraries with a modified linker script. Another approach would be to perform some fine-grained access pattern recognition of these headers.

V. Implementation

For implementing non-readable code, there are two different types of executables that have to be consider:

1) **Position-Dependent-Executables** (PDE). Code that has to loaded at a fixed address in memory. For PDE-executables it is hard to relocate parts of the program. The implementation for this scenario (see Sec. VI) uses the mirroring approach discussed earlier. This approach gives the process the illusion of a flat memory-model, while it actually is segmented.
2) **Position-Independent-Executables** (PIE). Programs that are composed entirely of Position-Independent-Code, which enables them to be executed regardless of where in memory it is loaded. PIE is often a requirement for ASLR [10]. The implementation for this scenario is presented in Sec. VII.

Both proof-of-concept defenses are implemented on x86 Linux. The implementation for both approaches consists of two parts: a kernel module and a user-space library.

For the defense designs presented in the following sections we have a few requirements that we want to adhere to:

1) **Enable on a per-process basis.** The defense mechanism presented here is of more interest to some applications, while other applications such as games, would relatively seldom benefit from such a defense mechanism.
2) **No kernel recompilation.** The defense should be possible to set up on an existing system with no down time.
3) **Do as much work as possible in user-space.** This follows naturally from the previous point. By using a user-space shared library to perform the address-space layout modifications, fewer lines of code are executed in kernel mode. Fewer modifications to the kernel greatly reduces the risk of accidentally introducing new exploits.

VI. Implementation: Shadow Address-Space (SAS)

In this implementation the available userland address space of 3 Gb is split into two segments of 1.5 Gb as per Fig. 4. The two segments are basically mirrors of each other; if a memory range is reserved in one segment, the same range is reserved in the other. The difference between the two segments lies in that executable mappings are only available in the *code*-segment, while the data is only loaded in the data segment.

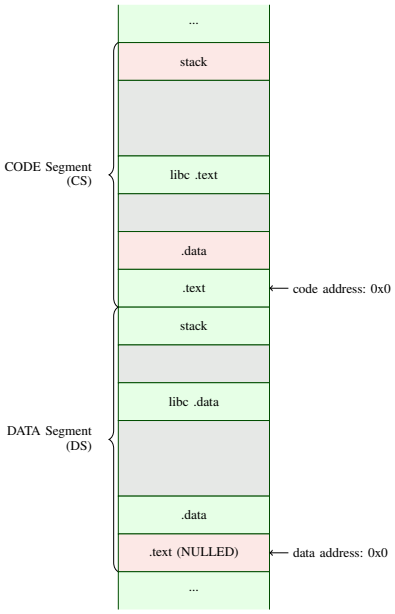
Thanks to segmentation, a code fetch at address *0x0* will be offset by the base of the code segment. In this implementation the *base* of the *code*-segment is *0x60000000*, which places it at an offset of 1.5 Gb from the start of the address space. By mirroring the memory layout across the segments, the SAS approach allows moving code sections in memory without changing the procedure pointers. In short the virtual address (before segmentation translation has been performed) remains unchanged, while the code is actually relocated. Because the virtual address stays the same, the pointers to the procedures do not need to be re-calculated. This reason makes this approach suitable for *position dependent executables*.

A. Address-space layout modifications

To achieve a mirroring of the address space, we have to be ensure that nothing is loaded at an address above the split (this threshold will hereafter be called SHADOW_CS_START). There are a few areas that need to be moved below this threshold:

1) **Shared libraries.** Usually the shared libraries are loaded at the top of the address space, just below the stack. By starting the executable in compatibility mode (*setarch i386 -L*) the old Linux address-space layout is used. This makes sure that dynamic libraries are loaded below SHADOW_CS_START.
   Alternatively, the *prelink* tool may be used to pre-link dynamic libraries at a specified address. The downside to pre-linking is that it disables ASLR for the pre-linked library. However, by regularly pre-linking the used libraries at a random address, the benefits of ASLR can be kept. Another approach of combining ASLR and prelink is discussed in [11].
2) **VDSO/ VVAR.** The VDSO/ VVAR segments are used for calling kernel space procedures in user space (e.g. system calls). Compatibility mode also moves the VDSO (virtual dynamically linked shared objects) section below SHADOW_CS_START.
3) **The Stack.** The stack is located at the top of the address space in Linux. When an executable is loaded, the *stack pointer* is initialized at the top of the address space. Environmental variables and command-line arguments are placed on the stack. By copying the *environmental variables* and *command-line arguments* to a new location and pointing the *stack pointer* to the new top of the stack, the stack can be moved freely. To reduce the complexity of moving the stack, it is moved before *libc* has been fully initialized. By intercepting the function *__libc_start_main* using *LD_PRELOAD*, the address

Fig. 4: **Shadow address-space layout.** The layout of the CODE and DATA segments are mirrors of each other. Only executable areas are mapped in the CODE segment, while the corresponding area in the DATA segment is unmapped.



location of the *environmental variables* can be obtained, and the stack can thus be moved.

B. Intercepting system calls

For achieving our XnR policy, all executable memory-mappings need to be placed at a mirrored area in the code segment. The mirroring of code mappings is achieved by intercepting the `mmap2` system-call using a kernel module. If the mapping has the flag *PROT_EXEC* it is mapped at the desired address plus SHADOW_CS_START.

When performing a code mapping, we have to make sure to return an address with the SHADOW_CS_START offset removed, so that the correct location will be accessed after segmentation address translation. Also, we have to ensure that mappings with the *MAP_FIXED* return an error if the address to be mapped at exceeds SHADOW_CS_START.

Furthermore, if `mmap` is called with the address 0x0, the Operating System decides where to put the mapping. Thus it also needs to be ensured that such a mapping is never placed above the threshold.

Besides `mmap`, the `mprotect` system-call may make memory areas executable. When *mprotect* is called with the *PROT_EXEC* flag, the corresponding area has to be re-mapped with an offset of SHADOW_CS_START. Likewise, when an executable area is made non-executable using `mprotect`, the area has to be moved with an offset of negative SHADOW_CS_START.

If an area is protected using both *PROT_READ* and *PROT_EXEC*, the area has to be mapped twice: once in the data segment and once in the code segment. This will of course add a certain overhead, since two calls to `mmap` are required.

```
if MAP_FIXED and addr > SHADOW_CS_START then
    return Error;
end
if PROT_EXEC then
    ret = mmap(addr + SHADOW_CS_START, ...);
    ret -= SHADOW_CS_START;
else
    ret = mmap(addr, ...);
end
return ret;
```

Algorithm 1: Intercepting mmap. When a mapping has the executable flag set, it is loaded in the upper part of the address space.

```
if PROT_EXEC and not marked as PROT_EXEC then
    map area to addr + SHADOW_CS_START;
end
if PROT_READ and not marked as PROT_READ then
    map area to addr;
end
return ret;
```

Algorithm 2: Intercepting mprotect. When an area is made executable, it is mapped in the shadow address space.

Furthermore, for a complete implementation, other mapping system calls such as `munmap` and `mremap` should be modified to follow the same guidelines as `mmap`. For `munmap` it is not known if a call un-maps an executable area, or a data area. However, since we use a mirrored address space, it should be safe to un-map the address range in both the *data* and the *code* section.

C. Specific obstacles

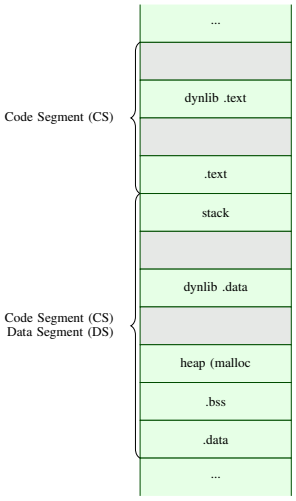
One obstacle for this implementation, is that a user process cannot alter its CS register (this would allow a process to enter ring-0), thus the register has to be set from the kernel. Using a kernel module a `proc`-file (called /proc/enable_xnr_cs) is created, which, when opened by a process, modifies the CS register of the calling process. A cleaner solution would be to set the correct register value on load-time, however, since the address-space layout of the program is altered using a user-space library, the protection can only be enabled after all such modifications have been performed.

D. Performance impact

For each `mmap` system-call some overhead is added by this approach. However, the added overhead is quite minimal, since it consist of a simple *if*-statement, which only adds an offset to a number.

However, with this approach some data has to be mapped two times, namely if it has to be both executable and readable. There is also a very particular scenario in which this approach would affect performance significantly. When using

Fig. 5: **Layout where the DATA segment is a subset of the CODE segment**. This makes a portion of the address-space executable but non-readable.



`mprotect` to alternate between setting something readable and execute-only. However, this scenario seems very unlikely.

VII. Implementation: Partially Disjoint Segments (PDS)

In this design the address space is modified to look as in Fig. 5. By setting the limit of the *data*-segment below the *code*-segment, an XnR-area is created at the top of the user-space. By placing all code in this XnR area, we ensure that the code is non-readable. To achieve a secure defence against ROP, all executable areas have to be in the XnR section.

A. Address-space layout modifications

To get a program running using this modified memory-layout we have to make sure that nothing that needs to be readable is mapped in the XnR area. As is the case for the *SAS*-approach, the stack has to be re-located. By placing the stack at a random location around address 0x50000000, it has enough room to grow, while not being in the way for other areas. Here, again, the VDSO/ VVAR areas have to be readable, thus starting the program in compatibility mode works fine. An upside of the PDS approach is that areas mapped below the XnR-section can still be executable without doing mirroring. Compared to the SAS-approach this makes things like moving the VDSO easier.

By moving the *.text* segment into the XnR section the program code becomes non-readable. This relocation is achieved by linking an executable with a linker script, which places the *.text* in the XnR section.

B. Relocating shared libraries

Of course, the above mentioned address-space modifications only ensure XnR for the program text. As stated earlier, the main way of exploiting ROP is by using procedures from `libc`. The executable parts of the shared libraries, thus has to be relocated. Achieving this task is not as trivial as it seems. When the dynamic loader loads a library, its mappings are placed in a contiguous range. That means that the *.data* and *.text* are placed next to each other in memory. By instrumenting the calls to procedures in libraries, the execution flow can be redirected to another address, thus making it possible to move procedures.

For achieving this instrumentation, the Dyninst library [12] is used. Dyninst allows us to modify the execution of a program while it is running. By redirecting calls from the *Procedure Linkage Table* (PLT), we can redirect library calls to any desired location. By mapping a copy of the code into the XnR section, while simultaneously unmapping the code from its standard location, the code is XnR protected. Since *x86* does not have relative addressing instructions for data, all references to data from the code should still be correct after relocation.

If a program is linked statically this instrumentation is not required, since the library code will be moved into the *.text* section of the ELF-binary when linking the program.

C. Performance impact

The approach taken for this implementation should have no run-time overhead during normal operation. When all relocations have been performed the program runs as usual. Due to the fact that dynamic library calls have to be instrumented, a certain overhead will incur from this instrumentation.

VIII. Limitations

One drawback of these implementations is that the they limit the size of the address space. For the shadow address-space this limitation is more apparent than for the disjoint segment approach. For the disjoint segment approach the *XNR_THRESHOLD* could be adjusted on a per-process basis, keeping a larger size of the address space usable. Please note that the SAS-approach does not increase RAM usage, since pages that are mapped twice, only have to reside in physical memory once.

Furthermore, there is the problem of self-modifying programs. Self-modifying programs need code that is both writable and executable. In the case of the SAS-implementation the modifiable code would be loaded twice: once in the data segment, once in the code segment, to make it both writable and executable. When data in the *data segment* is modified the changes have to be mirrored in the *code segment*. This could be done using some kind of instrumentation. It has, however, been decided to not support self-modifying programs. Mainly, because they are not that common and, additionally, they make it possible to perform standard buffer-overflow attacks (since $W \oplus X$ has to be disabled).

Also, this defense only applies to memory-disclosure attacks. If an address of a function can be obtained by an attacker, using for example a leaking pointer, a gadget can still be assembled. However, since this approach only reveals the start location of procedures, construction of small, fine-grained

gadgets (e.g. POP-return) is not possible, thus the range of attacks are far more limited.

Finally, this implementation is only possible on 32-bits x86 architecture. In 2006 Intel decided to drop support for segmentation for their current line of processors. Because of this reason, the defense mechanisms presented in this paper are not available on x86-64.

IX. Related work

A comparable defense against JIT-ROP has been presented by Backes *& al* [8]. Their approach is to emulate XnR by marking pages containing code as XnR. The marking is done by intercepting page faults and keeping a cache of which pages should be execute-only. By marking them absent in the MMU, each time such a page is accessed, the MMU generates an interrupt. This interrupt is then caught, whereafter it is decided if the access is legitimate or not. Backes & al claim a overhead of merely 2.2% for Linux. They achieve this low overhead manly due to the fact that functions calling each other reside close to each other in memory (thus likely on the same page). If this approach were used in combination with a more fine-grained *ASLR* chances are that the performance would be much worse. By moving procedures to random areas in memory it is likely that procedures within a library are scattered across different pages. Furthermore, the defense by Backet *& al* does not protect against access to the currently available pages, which in their case is two or more for the sake of performance. Arguably, it is possible that two pages (or about 8 KB) could contain some possible gadgets, thus making the defense vulnerable.

Another approach, besides ASLR, that tries to mitigate *ROP* attacks is presented by Barrantes in [13]. They use an emulated randomized instruction set to make it harder to scan the memory for known gadgets. Naturally there is some performance overhead involved with emulating an instruction set.

A diversification defense that mitigates ROP gadget chains by means of altering the execution path is presented in [14]. By having two differently diversified versions (using fine-grained ASLR) of a program loaded in memory, the execution path can be altered. By, figuratively, constantly flipping a coin on which version of the code to use, assembled ROP gadget chains cannot execute reliably.

Approaches such as *Redactor* [15] combine different techniques to protect against ROP. First, they achieve execute-only memory by using virtualization capabilities on x86. Secondly, they implement code diversification on compile-time. Finally, they hide code pointers on compilation using `jmp` trampolines, making it hard to find the actual address of procedures. The overhead incurred by *Redactor* is 6.4%.

A conceptually very similar implementation to the SAS-implementation presented in this paper is used by PaX [16]. Despite their goal being to enable non-executable data, rather than XnR, their implementation fully supports XnR by marking a segment execute-only. The overhead claimed by *PaX SEGMEXEC* is about 0.7% [3].

Likewise, Red Hat's Exec Shield contains a security measure to emulate $W \oplus X$ using segmentation that is very similar to the *PDS*-approach [17]. By setting the limit of the code segment below the limit of the data segment, an area is created that contains data that will be non-executable. Exec Shield requires all program code to be loaded below this limit. The downside to this approach, as is with the PDS-approach presented in this paper, is that programs have to be re-linked if they are not position independent.

X. Conclusion

In this paper we have demonstrated the need for non-readable code. If code is readable, it is possible to assemble *ROP* gadget chains by making use of a buffer overflow. By scanning the memory using a memory disclosure exploit, such chains can be assembled *just-in-time* for a target, without knowing exactly where procedures are loaded in memory. If executable *code* is made non-readable, scanning the memory for gadgets becomes impossible. Thus, in combination with other diversification defenses, such as*ASLR*, non-readable code constitutes a strong defense against *JIT-ROP*.

By modifying the address space of a process, it is possible to achieve an efficient implementation of XnR using segmentation memory-translation hardware. We have presented two such defense-approaches: one for all executables, and one exclusively for *position independent executables*.

The first approach achieves XnR by using separate segments for code, and data, while giving the illusion of a flat memory-model to the process at the cost of half of the address-space.

The second approach limits the *data*-segment, thus creating an XnR section at the top of the address space. The benefit of this approach is that the full address-space can still be used.

XI. Future Work

Despite segmentation being deprecated, it should still be possible to implement a defense similar in nature to the PDS implementation on new Intel hardware. By using new memory bounds-checking instructions (MPX) introduced by Intel it is possible to check that a memory access is within certain bounds. A similar split between code and data as in the PDS implementation could be made, thus making a portion of the address space executable, but not readable. By inserting an MPX call before every memory read, it possible to specify an upper bound for all memory read instructions, effectively creating an XnR section.

The downside to this approach is the overhead. For each memory read, an extra instruction has to be inserted. Benchmarks for MPX show that there is an enormous overhead. However, when looking at the generated code when enabling MPX bounds-checking in GCC, it is obvious that a lot of overhead is due to managing the bounds registers. Since the approach presented here only requires one bound-check, it could be safe to assume that the performance should be reasonable (why else would Intel introduce such an instruction?).

[3] This claim can be found here: https://wiki.ubuntu.com/PaXvExecShield. However no experimental data is available.

REFERENCES

[1] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," 2005.

[2] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[3] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 298–307.

[4] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino, "Marlin: A fine grained randomization approach to defend against rop attacks," in *Network and System Security*. Springer, 2013, pp. 293–306.

[5] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *USENIX Security Symposium*, 2014.

[6] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[7] P. S. Ramu Ramakesavan, Dan Zimmerman, "Intel memory protection extensions (intel mpx) enabling guide," Apr. 2015. [Online]. Available: https://software.intel.com/sites/default/files/managed/9d/f6/Intel_MPX_EnablingGuide.pdf

[8] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1342–1353.

[9] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Prentice Hall Press, 2014.

[10] U. Drepper, "Security enhancements in redhat enterprise linux (beside selinux)," *Retrieved November*, vol. 15, p. 2009, 2005.

[11] H. Yoon, C. Min, and Y. I. Eom, "Dynamic-prelink: An enhanced prelinking mechanism without modifying shared libraries."

[12] "Dyninst: An application program interface (api) for runtime code generation," *Online, http://www. dyninst. org*.

[13] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 281–289.

[14] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," *Proc. 22nd Network and Distributed Systems Security Sym.(NDSS)*, 2015.

[15] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *IEEE Symposium on Security and Privacy, S&P*, vol. 15, 2015.

[16] (2006, Oct.) Pax segmexec. https://pax.grsecurity.net/docs/segmexec.txt.

[17] A. van de Ven, "New security enhancements in red hat enterprise linux v. 3, update 3," *Raleigh, North Carolina, USA: Red Hat*, 2004.