Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/jisa

ROPocop — Dynamic mitigation of code-reuse attacks

Andreas Follner [*], *Eric Bodden*

Secure Software Engineering Group, Technische Universität Darmstadt, Rheinstr. 75, 64295 Darmstadt, Germany

ARTICLE INFO

Article history:
Available online 22 February 2016

Keywords:
Buffer overflow
Return-oriented programming
Code-reuse attack
System security
Exploit mitigation
Dynamic binary instrumentation

ABSTRACT

Control-flow attacks, usually achieved by exploiting a buffer-overflow vulnerability, have been a serious threat to system security for over fifteen years. Researchers have answered the threat with various mitigation techniques; but nevertheless, new exploits that successfully bypass these technologies still appear on a regular basis.

In this paper, we propose ROPocop, a novel approach for detecting and preventing the execution of injected code and for mitigating code-reuse attacks such as return-oriented programming (RoP). ROPocop uses dynamic binary instrumentation, requiring neither access to source code nor debug symbols or changes to the operating system. It mitigates attacks both by monitoring the program counter at potentially dangerous points and by detecting suspicious program flows.

We have implemented ROPocop for Windows x86 using PIN, a dynamic program instrumentation framework from Intel. Benchmarks using the SPEC CPU2006 suite show an average overhead of 2.4×, which is comparable to similar approaches, which give weaker guarantees. Real-world applications show only an initially noticeable input lag and no stutter. In our evaluation our tool successfully detected all 11 of the latest real-world code-reuse exploits, with no false alarms. Therefore, despite the overhead, it is a viable, temporary solution to secure critical systems against exploits if a vendor patch is not yet available.

1. Introduction

Attacks that aim at manipulating a program's control flow, often through a buffer overflow vulnerability, are still one of the biggest threats to software written in unsafe languages like C or C++ (Bubinas, 2013). If successfully exploited, control-flow attacks can allow an adversary to execute arbitrary code. In the early 2000s, operating-system developers started adding mitigation techniques into their software. To this day, new techniques are added on a regular basis; however, while they make successful and reliable exploitation much more difficult, they can be bypassed.

Contests like, e.g., pwn2own (Gunn, 2014) continuously show that current mitigation techniques are insufficient when it comes to protecting applications, and that more comprehensive methods are required. Currently, the most widely used attack technique, and an essential part of virtually every exploit, is RoP (Roemer et al., 2012), where instead of injecting new code, an attacker pieces together short code fragments, which already

[*] *Corresponding author.* Secure Software Engineering Group, Technische Universität Darmstadt, Rheinstr. 75, 64295 Darmstadt, Germany. Tel.: +49 6151 869342; fax: +49 6151 869127.

E-mail address: andreas.follner@cased.de (A. Follner).

http://dx.doi.org/10.1016/j.jisa.2016.01.002
2214-2126/© 2016 Elsevier Ltd. All rights reserved.

exist in memory. Recently proposed solutions against such attacks mostly built on CFI (Pappas et al., 2013; Zhang et al., 2013; Zhang and Sekar, 2013) seemed effective, but have been shown to be bypassable (Davi et al., 2014; Göktaş et al., 2014). Section 2 elaborates on these issues in detail.

To battle current exploitation mechanisms we propose ROPocop, a novel tool that mitigates control-flow attacks for x86 Windows binaries using two novel techniques: AntiCRA and DEP+. AntiCRA greatly reduces the risk of successful code-reuse attacks by detecting an unusually high rate of successive indirect branches during the execution of unusually short basic blocks. As different programs can exhibit very different behavior with regard to that aspect, using the same threshold for every program is suboptimal. Therefore, ROPocop comes with a learning mode, which runs ahead of time and determines appropriate thresholds, which can be adopted by the user. However, we do also provide default thresholds which work very well in practice and for a large selection of programs, as our evaluation shows.

Our second contribution, DEP+, implements a variant of a non-executable stack through dynamic binary instrumentation. DEP+ assumes that all code has to reside within a program image, i.e., the .text section of any PE file. This is very similar to DEP (Andersen and Abella, 2004); however, DEP+ cannot be disabled through API calls, thereby eliminating a large class of exploits that are based on such calls. DEP+ enforces all memory to be non-executable, except for the parts to which images are loaded. To this end, DEP+ monitors the loading and unloading of images, checking after each indirect branch whether the program counterpoints outside the known images.

We have implemented ROPocop for Windows x86 using PIN (Luk et al., 2005), a freely-available dynamic program instrumentation framework from Intel. ROPocop requires no access to source code or root privileges, nor debug symbols or changes to the operating system. Measurements using the artificial SPEC CPU2006 suite show an average overhead of 2.4×. More importantly, experiments on real-world applications show only an initially noticeable input lag (caused by the initial dynamic instrumentation) and no stutter. Our evaluation using 11 of the latest code-reuse exploits shows that our tool successfully prevents all code-injection attacks and code-reuse attacks from succeeding, even a highly sophisticated attack that relies solely on code reuse (Li and Szor, 2013). Our envisioned usage of ROPocop is to use it as a last line of defense against exploitation of critical systems, e.g., when a severe vulnerability has been discovered but no patch is available.

To summarize, this work makes the following original contributions:

- AntiCRA, a tunable heuristic detection of code-reuse-attacks like RoP and JoP,
- DEP+, a comparatively fast and very robust implementation of a non-executable stack,
- ROPocop, a dynamic instrumentation tool based on PIN which detects various kinds of control-flow attacks using the above techniques, and
- an empirical evaluation showing that ROPocop ‘s mitigation approach is highly effective and shows tolerable runtime overheads.

We make ROPocop available online as open source, along with all our experimental data (https://sites.google.com/site/ropocopresearch/).

2. Current situation

Exploiting vulnerabilities with the goal to manipulate the program flow was relatively trivial on Windows until the early 2000s, when Microsoft began adapting mitigation techniques. In the simplest cases, an attack widely known as *stack smashing* (One, 1996) could be used. Such an attack would leverage unbounded functions, such as strcpy, to write beyond the allocated memory of a buffer. Attackers could thus overwrite the function's stored return address on the stack with an address that points to injected code, which the program will execute after the next return.

To defend against such code injection attacks, Microsoft implemented *Data Execution Prevention* (DEP) (Microsoft), which makes use of a processor's NX (no execute) bit. DEP marks pages which contain data as non-executable, causing a hardware-level exception if execution from within such a page is attempted. This successfully prevents attacks that attempt to execute injected code.

Nevertheless, attackers can bypass DEP in various ways. At present, the most widely used technique is called return-oriented programming (Shacham, 2007). When utilizing RoP, an attacker does not inject any code but instead uses existing code fragments (*gadgets*), which all end with a return instruction. In other words, instead of injecting code, the attacker injects the addresses of the gadgets he wants to execute. On x86, return works by popping an address off the stack into the register EIP and then jumping to that address. By crafting a stack filled with a sequence of gadget addresses, the attacker can execute sequences of gadgets, with the return instruction at the end of each gadget transferring the program flow to the next gadget. Jump-oriented programming (JoP) (Bletsch et al., 2011; Checkoway et al., 2010; Min et al., 2012) is based on the same basic concept as RoP, but uses jmp instructions to transfer control flow to the next gadget. In the following, we refer to both RoP and JoP attacks as *code-reuse attacks*.

The success of code-reuse attacks depends on the availability of useful gadgets on the target platform and the complexity of the code the attacker wants to run. In practice, however, most systems are vulnerable to such code-reuse attacks. Furthermore, RoP attacks are relatively complex to stage, which is why most attacks of this kind do not resort to pure RoP, but rather implement a two-staged approach. The first stage uses RoP to call a Windows API function like `VirtualProtect` (see below) which marks a certain memory region as executable, effectively bypassing DEP. This is followed by the second stage, running code previously injected into that memory region, which can then be executed as normal. Code-reuse attacks work reliably if the memory layout of an application is highly deterministic because an attacker can hard-code the addresses of gadgets directly into the exploit. To mitigate this, Microsoft introduced randomness in the form of ASLR (Howard et al., 2010). ASLR randomizes the order in which images are loaded into the virtual address space and adds pseudo-random offsets to their base addresses. This makes it very difficult for an attacker to predict the memory locations of the required gadgets on the target system.

2.1. *DEP weaknesses*

Whether or not a program is protected by DEP depends on the compiler setting the `NX_COMPAT` flag in the header of the program's main executable. This flag may be left unset due to a number of reasons, including unsafe compiler defaults and program incompatibility. Thus, the opt-in nature of DEP may render its benefits void. If it is enabled, an attacker can only bypass it through previously discussed code-reuse attacks. Such attacks succeed without ever running code from non-executable pages, which is why DEP cannot protect against them. The previously mentioned two-staged attack, however, is worth explaining in more detail, since it allows injected code to be run, effectively bypassing DEP. Such attacks work because Windows exposes certain functions to change the permissions of a page (e.g. `VirtualProtect`) or allocate new pages with specific permissions (e.g. `VirtualAlloc`), which are passed as parameters. These APIs are necessary so programs that generate and execute code at runtime (e.g., browsers) can still be compatible with DEP. Such programs set the `NX_COMPAT` flag, hence are protected by DEP; however, when they need to use JIT compilation, they can use the previously mentioned APIs to allocate executable memory. Which leads to the issue, that an attacker can also invoke those APIs with parameters of their choice, i.e., even if the original program does not invoke, e.g., `VirtualProtect` with read/write/execute priviliges, an attacker can.

2.2. *ASLR weaknesses*

Like DEP, ASLR is not necessarily enabled, which depends on a flag called `DYNAMIC_BASE`. However, unlike DEP, it is not either on or off for the whole process. The operating system is able to handle processes composed of a mixture of ASLR-enabled and disabled images, which simply means that some images will get rebased and others will not. Apart from legacy libraries which were compiled before ASLR existed, a library might not support ASLR because parts of a program use hard-coded jump addresses within that library. Bypassing ASLR appears to be difficult in practice, with no currently-known generic attack. The work by Shacham et al. (2004) relies on brute force, which only works if the vulnerable application does not crash when an access violation occurs. *Partial overwrites* (Durden, 2002) overwrite only the last two bytes of an address on the stack. Because only the first two bytes get randomized, this attack does not require knowledge of the randomness introduced by ASLR. This gives an attacker a range of at most 4096 bytes of instructions. Durden presents information leaks as a way of gathering information about the memory layout of an ASLR-protected application (Durden). Hund et al. (2013) propose a timing-based side channel attack that can break kernel space ASLR within minutes, given that an attacker knows the hardware of the attacked system. However, most current exploits do not have to use such techniques, and instead can rely on the presence of some non-ASLR images on the target platform. Such images, however, are still very common on current systems, which is why many attacks still succeed.

2.3. *Attacker model*

We assume a relatively strong attacker, who is able to bypass DEP, ASLR, and other mitigation techniques which are currently part of Windows. This is, for a determined attacker, a realistic assumption. We even go one step further and allow for pure RoP and JoP attacks, which do not have to call `VirtualProtect` or `VirtualAlloc` but instead rely solely on code reuse by chaining gadgets and existing API calls together in such a way, that the attacker can achieve her goal without injecting any code. Such pure code-reuse attacks are still rarely found in the wild, but we expect them to increase due to the work that is being done on detecting two-staged attacks. One known example is a pure RoP attack on Adobe Reader (Li and Szor, 2013).

3. AntiCRA

When designing AntiCRA, we manually analyzed RoP and JoP exploits by looking at the gadgets they use and their properties. We found that the exploits share properties which are unusual and typically not present in a normal program's execution. Based on these observations, we built a heuristic which monitors the following two properties:

Indirect branches

Code-reuse attacks consist of gadgets which all end in an indirect branch. We analyzed benchmarks as well as real-world applications like Adobe Reader, VLC, Microsoft Office, Open Office (the complete list can be found in Table 1) and found that executing a very high number of consecutive indirect branches is rather unusual. The highest number of subsequent indirect branches we found during our experiments was 47 (in Microsoft Word), but only 8 of the 35 programs execute 15 or more subsequent indirect branches.

Average length of basic blocks

To reduce side-effects on other registers, the stack, or flags, exploit developers try to use gadgets that are as short as possible. Therefore, at least for contemporary approaches, gadgets can be considered basic blocks with very few instructions. As with indirect branches, we analyzed program behavior of legitimate programs and found that the average number of instructions over a sliding window of 10 basic blocks did not drop below 2.33. Our experiments showed that making the window smaller resulted in an increase of false positives, while increasing the window resulted in missed attacks. We also found an interesting correlation between this and the previous property: the more consecutive indirect branches, the longer the corresponding basic blocks. We make use of this knowledge in the next paragraph, when we try to find default parameters which work for a wide set of applications.

As previously mentioned, since programs can exhibit varying characteristics regarding these two properties, ROPocop first

Table 1 – Analysis of exploits and programs. We assigned a unique ID to every exploit and program we investigated. The last two columns show our metrics for AntiCRA, i.e., the highest number of indirect branches taken and the average basic block length. Bold numbers show that AntiCRA was triggered, due to an exceeded threshold. [*Not computed by AntiCRA due to low number of indirect branches (<15). **Data are based on the analysis by Li and Szor (2013)].

ID	Name	Indirect branches, threshold: 35	Average BB-Length, threshold: 2.25
E01	ASX to MP3 Converter v3.1.2.1 SEH Exploit (Multiple OS, DEP and ASLR Bypass)	**>50**	>2.25
E02	BlazeDVD 5.1 Stack Buffer Overflow With ASLR/DEP Bypass	20	**1.9**
E03	BlazeDVD 6.1 PLF Exploit DEP/ASLR Bypass	16	**2**
E04	DVD X Player 5.5.0 Pro/Standard version Universal Exploit, DEP + ASLR Bypass	17	**2**
E05	DVD X Player 5.5 Pro (SEH DEP + ASLR Bypass) Exploit	13	(2.2)*
E06	ProSSHD 1.2 remote post-auth exploit (w/ ASLR and DEP bypass)	**43**	**2**
E07	RM Downloader 3.1.3 Local SEH Exploit (Win7 ASLR and DEP Bypass)	**49**	>2.25
E08	The KMPlayer 3.0.0.1440 .mp3 Buffer Overflow Exploit (Win7 + ASLR bypass mod)	**46**	>2.25
E09	UFO: Alien Invasion v2.2.1 BoF Exploit (Win7 ASLR and DEP Bypass)	**>50**	>2.25
E10	Winamp v5.572 Local BoF Exploit (Win7 ASLR and DEP Bypass)	**>50**	**2**
E11	Adobe Reader 11.0.01 "Number of the Beast" (ASLR, DEP, Sandbox bypass, pure RoP)**	**>50**	—
E12	QQ PLAYER PICT PnSize Buffer Overflow WIN7 DEP ASLR BYPASS	11	(2)*
A01	Daemon Tools v. 4.47.1.0333	40	4.97
A02	Microsoft Word 2010 v. 14.0.07140.5002	47	4.14
A03	Microsoft Excel 2010 v. 14.0.07140.5002	28	4.1
A04	Microsoft PowerPoint v. 14.0.07140.5002	25	4.14
A05	Adobe Acrobat Pro v. 9.0	29	2.33
A06	Windows Media Player v. 12.0.7601.18150	11	na*
A07	cmd.exe (on Windows 7 Pro SP1 64 bit v. 6.1.7601)	5	na*
A08	calc.exe (on Windows 7 Pro SP1 64 bit v. 6.1.7601)	6	na*
A09	mspaint.exe (on Windows 7 Pro SP1 64 bit v. 6.1.7601)	13	na*
A10	taskmgr.exe (on Windows 7 Pro SP1 64 bit v. 6.1.7601)	9	na*
A11	VLC v. 2.0.8	9	na*
A12	Irfanview v. 4.3.3	12	na*
A13	Notepad++ v. 6.1.4	11	na*
A14	Filezilla v. 3.5.3	7	na*
A15	Open Office Writer v. 4.0.1	7	na*
A16	Open Office Impress v. 4.0.1	7	na*
A17	Open Office Calc v. 4.0.1	8	na*
B01	SPEC CPU 2006 — 400	4	na*
B02	SPEC CPU 2006 — 401	3	na*
B03	SPEC CPU 2006 — 403	6	na*
B04	SPEC CPU 2006 — 429	3	na*
B05	SPEC CPU 2006 — 433	3	na*
B06	SPEC CPU 2006 — 444	4	na*
B07	SPEC CPU 2006 — 445	5	na*
B08	SPEC CPU 2006 — 447	7	na*
B09	SPEC CPU 2006 — 450	8	na*
B10	SPEC CPU 2006 — 453	6	na*
B11	SPEC CPU 2006 — 456	3	na*
B12	SPEC CPU 2006 — 458	3	na*
B13	SPEC CPU 2006 — 464	31	4
B14	SPEC CPU 2006 — 470	4	na*
B15	SPEC CPU 2006 — 471	15	3.91
B16	SPEC CPU 2006 — 473	3	na*
B17	SPEC CPU 2006 — 482	9	na*
B18	SPEC CPU 2006 — 483	17	4

runs in learning mode. This requires nothing from the user but simply using the program she wants to protect as usual, while in the background, ROPocop observes the program flow and determines appropriate thresholds for these two properties. This, of course, leads only to limited coverage; however, for our approach high coverage is not required. Exploiting a buffer overflow requires some sort of input, generally provided by the attacker as a file that has to be opened by the victim and is then processed by the vulnerable program. Thus, a user working with the program might not cover all possible paths, but it covers the important paths which lead to exploitation.

As expressed earlier, we recommend setting individual thresholds for different programs, but at the same time we evaluated whether it is possible to provide default values which cover as many programs as possible. After analyzing our test set of benign applications, by running the learning mode and using the programs in our sample set (e.g., opening various media files using VLC, opening various PDF files with Adobe Reader, working with Microsoft Word, etc.), we set the following thresholds: 35 subsequent indirect branches and an average basic block length of 2.25 or lower; as described earlier, we found a correlation that larger numbers of subsequent basic blocks

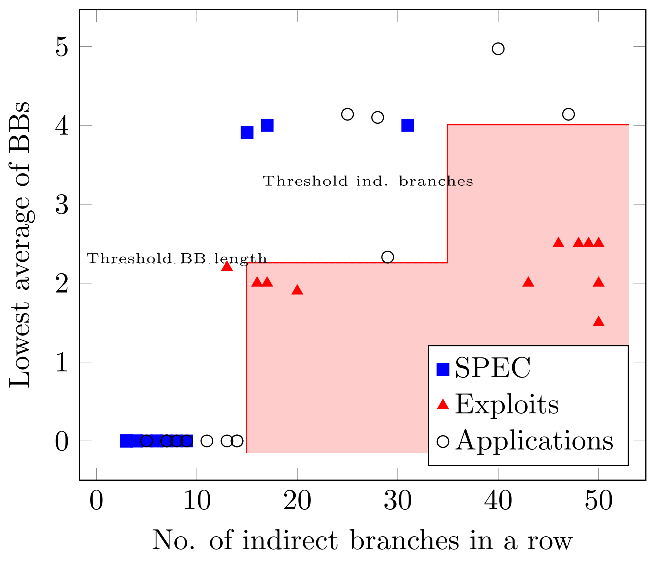

Fig. 1 – Analysis of the number of indirect branches in a row and the lowest average basic block length of our test set.

also means longer basic blocks. Therefore we added another threshold; 36–50 subsequent indirect branches and an average basic block length of 4 or lower. AntiCRA signals an exploitation attempt if one of the two bounds is violated or if, at any point, more than 50 subsequent indirect branches are executed. While our sample set of benign applications may not be large enough to make a claim, that these suggested thresholds hold for all programs, they do hold for all programs in our set, which includes some of the most exploited applications. Therefore, they serve as an excellent starting point for fine-tuning, should it be required. Since we included many programs that are often found and exploited in business environments (e.g., Word, Excel, Adobe Reader), ROPocop can be deployed immediately without the need to fine-tune thresholds.

To increase performance and make the algorithm less prone to false positives, calculating averages starts only after we have collected 15 basic-block lengths, i.e., the first computed average is available only after 15 subsequent indirect branches. This prevents false alarms based on short sequences of short basic blocks, whose sample size is otherwise not significant enough. While this allows for attacks requiring less than 15 gadgets to go undetected by AntiCRA, it decreases false positives because in our experiments we discovered sequences of three to seven consecutive indirect branches with small basic block lengths which would trigger a false alarm. However, these parameters may be changed by the user. Attacks that use less than 15 gadgets are very likely non-pure RoP attacks and will therefore be detected by DEP+ (Section 4). Algorithm 3 summarizes AntiCRA in a formal way. Fig. 1 (in Section 5) shows how the two thresholds form a (shaded) area in a two-dimensional plain. If an execution falls into the shaded area, then AntiCRA will signal it as malicious. The figure also summarizes the results of our empirical evaluation and will be explained in more detail later.

Input: *bbl:* a basic block, delivered automatically by PIN, *thrStartAvgCalc*: the number of consecutive indirect branches after which the calculation of average basic block length starts (default: 15), *thrAlarm*: the threshold for the average basic block length (default: 2.33)
Output: *state:* a flag that indicates that a RoP attack is probably in progress

```
Input: bbl: a basic block, delivered automatically by PIN,
       thrStartAvgCalc: the number of consecutive indirect branches
       after which the calculation of average basic block length starts
       (default: 15), thrAlarm: the threshold for the average basic
       block length (default: 2.33
Output: state: a flag that indicates that a RoP attack is probably in
        progress
state ← noAlarm
cntIndBranch ← 0
avg ← 0
if bbl was reached through indirect branch then
    cntIndBranch ← cntIndBranch + 1;
    log size of bbl;
    if cntIndBranch >thrStartAvgCalc then
        avg ← average length of the last 10 bbl;
        if avg >thrAlarm then
            state ← alarm;
        end
    end
else
    cntIndBranch ← 0;
end
```

Algorithm 1: Algorithm for AntiCRA. We use PIN to continuously monitor basic blocks at runtime and use them as input for our algorithm. Once enough basic block lengths have been collected, i.e., *thrStartAvgCalc* is exceeded, calculation of average basic block length starts. If this average falls below *thrAlarm*, an alarm is raised.

3.1. *Impact on current and future exploits*

For a code-reuse attack to circumvent AntiCRA, it must not use more than 34/49 consecutive indirect branches. If this is possible at all depends on the availability of gadgets, which varies between programs based on what libraries are loaded and whether or not ASLR is being employed. Furthermore, the average number of instructions in the gadgets used must never fall below 2.25/3.5. Combined, these restrictions make it very difficult for an attacker to create a pure RoP or JoP payload. Attackers could attempt to raise the average number of instructions per gadget by inserting longer gadgets. But longer gadgets usually have unwanted side-effects, like manipulating other registers that hold important data, or the stack, or modifying flags. Furthermore, since the total number of gadgets is limited to 34/49, inserting long gadgets whose side effects are irrelevant just for the sake of increasing the average wastes precious slots for useful gadgets. To bypass AntiCRA, an attacker would have to try and insert direct branches, but, due to limited availability and side-effects, e.g., potentially losing control over the program counter, this is anything but trivial. Furthermore, we know of no gadget compiler that would support direct branches at this point and have not found exploits that use gadgets that incorporate direct branches. Depending on the program it might still be possible, but, as previously mentioned, our goal is to break current exploits and make the development of new code-reuse exploits significantly more difficult, which AntiCRA certainly achieves. Long NOP gadgets, as proposed by Davi et al. (2014), could potentially be used to artificially increase the average basic block length hence bypass AntiCRA; however, it takes five gadgets to restore the stack and registers to their original form. Therefore, precious space has to be wasted and such an attempt will most likely exceed any sensible threshold. Furthermore, the authors state that finding such a gadget was “a non-trivial task that required painstaking analyses and a stroke of luck”, so for some programs this technique might not be possible at all.

3.2. *Limitations*

Due to its heuristic nature, false positives as well as false negatives are possible. As we show in this work, however, in practice the heuristic seems effective enough to go without any false decisions, at least in our benchmark set. Furthermore, under circumstances very favorable to an attacker it might be possible to create a two-staged exploit that disables DEP using fewer than 15 gadgets and then runs a regular payload. This would not be detected by AntiCRA and motivates the need for reliably non-executable data sections, which we enforce using DEP+ (Section 4).

4. DEP+

DEP+ is based on the same concept as DEP, i.e., the premise that data should not be executable. DEP+ thus monitors the loading and unloading of images and creates a virtual memory map based on this information. All virtual memory space where no image is mapped is considered to hold potentially malicious data, since Windows can allocate stacks or heaps in these areas. To enforce that the instruction pointer EIP never points outside an image, DEP+ checks the register’s value after each indirect branch, i.e., after each return, indirect call, and indirect jump. Opposed to DEP, DEP+ cannot be bypassed through API calls such as `VirtualProtect`.

4.1. *Implementation details*

PIN’s `IMG_AddInstrumentFunction` as well as `IMG_AddUnloadFunction` are used to monitor the loading and unloading of images. When an image is loaded, DEP+ stores its start and end address; if the same image is unloaded at runtime, this information is removed. This approach results in a virtual-memory map that distinguishes only between images and non-images, i.e., code regions and data regions. DEP+ treats these data regions as space for potentially malicious data, hence does not allow EIP to point into it. To do so, DEP+ checks after any indirect branch is taken, but before it is executed, if the instruction pointer points inside any of the data regions. Algorithm 2 summarizes DEP+ in a formal way.

Input: *p*: a program, *mmap*: a map of virtual memory that contains start and end addresses of loaded images and allows calculating start and end addresses of data regions
Output: *state*: a flag that indicates that code is executed from outside an image

```
Input: p: a program, mmap: a map of virtual memory that contains
       start and end addresses of loaded images and allows calculating
       start and end addresses of data regions
Output: state: a flag that indicates that code is executed from outside
        an image
state ← noAlarm
if instruction pointer points inside a data region in mmap then
  | state ← alarm
end
```

Algorithm 2: Algorithm for DEP+ (without performance optimizations). We use PIN to instrument the target program so the algorithm is called after an indirect branch is taken, but before the instruction is executed. It checks whether the instruction pointer points into a data region, i.e., outside all loaded images.

The reason DEP+ checks if EIP points inside data regions instead of checking if EIP points inside a loaded image is due to performance: some programs load 30 or more libraries, which means that there can be an equally high number of code regions that need to be checked. As we found, checking each of those regions after each indirect branch can incur a significant performance penalty. To increase performance, we thus make use of the fact that Windows’ memory management is relatively deterministic. Images, in general, tend to be loaded at very high addresses, around `0x60000000` and higher, while stacks and heaps reside at low addresses and new ones are allocated towards increasingly higher addresses. Depending on the memory usage of a process, it is generally valid to assume that stacks and heaps, where an attacker would inject his payload, reside below most images.

DEP+ makes use of this knowledge by not checking if EIP points inside any of the loaded images but instead checking if EIP points inside data regions, where heaps and stacks are located, which results in a much lower number of necessary checks. To this end, DEP+ monitors a program’s heap and stack sizes to dynamically increase or decrease the number of data regions that need to be taken into account. We implement this by probing memory usage every 10th time a function that allocates or de-allocates memory is called and multiply the reported usage by 1.3 to have a large enough safety margin. Only data regions within that memory area will be checked.

This is, of course, a heuristic, which trades security for performance, but as our evaluation in Section 5 shows, the heuristic helps DEP+ to bring the checks down to a minimum while still recognizing all tested attacks. Furthermore, our experiments using benchmarks and real-world programs have shown that memory allocations are done in many small steps, hence probing memory usage in short intervals and adding a safety margin of 30% has never failed to correctly detect the necessary number of regions which have to be checked. For the heuristic to fail and be exploitable, it would take one single memory allocation the size of about 30% of the current memory usage, a vulnerable function which uses this memory and an instruction which redirects program flow into this memory before our algorithm checks memory usage again. Since an attacker has limited influence on these preconditions, we accept the risk that our heuristic might fail under rare conditions, e.g., depending on the underlying vulnerability and the environment, a heap spray in a browser might enable an attacker to bypass DEP+. Reliably exploiting such circumstances in a multi-threaded program, however, would be even more difficult due to their highly non-deterministic nature and the fact that the attacker does not know when the next memory usage calculation will be triggered by DEP+.

4.2. *Comparison to DEP*

The original shortcomings of DEP are that it may not be enabled at all, or that it can be bypassed by both pure code-reuse attacks and by code-reuse attacks that invoke `VirtualProtect`, etc., to disable DEP. DEP+ improves over DEP in that it prevents the execution of injected code by enforcing non-executable data regions *even* for processes that run with regular DEP disabled. In particular, DEP+ cannot bypassed by calls to `VirtualProtect` and its siblings, as such calls have no effect on DEP+. A similar result could be achieved by hooking said functions and simply not executing them. However, userland hooks can be bypassed easily (Butler, 2004) and kernel hooks require administrator privileges, hence make deploying our solution more complicated. Furthermore, as Section 5 shows, the overhead introduced by DEP+ is negligible.

4.3. *Limitations*

Processes which rely on the ability to execute code from outside images, e.g., processes which generate code at runtime or incorporate self-modifying code, are not compatible with DEP+. Such a process is not compatible with DEP either, unless it uses the `VirtualProtect` API, etc., to disable DEP for memory regions with generated code. Since it is difficult to detect whether a call to the API usually abused to bypass DEP by an attacker is legitimate, i.e., originating from the program itself, we decided against supporting such calls. This results in a strong increase in security, at the drawback of slightly reduced compatibility with mostly older software.

Like DEP, DEP+ cannot detect and thus not prevent the exploitation of the vulnerability itself, e.g., the overwriting of data on the stack due to a buffer overflow. Therefore, non-control data attacks (Chen et al., 2005) or information leakages are still possible. Furthermore, DEP+ does not prevent *pure* code-reuse attacks, motivating the need for AntiCRA (Section 3).

5. Evaluation

Our implementation is highly modular, so that one may deploy AntiCRA or DEP+ independently as well as in combination. Running both of them, however, strongly increases security, in a similar fashion as running with DEP and ASLR.

In this chapter we evaluate AntiCRA and DEP+ by addressing the following research questions:

RQ1: How effectively does AntiCRA detect pure code-reuse payloads?
RQ2: How effectively does AntiCRA detect two-staged RoP payloads?
RQ3: How effectively does DEP+ detect code-injection attacks?
RQ4: What is the performance overhead of AntiCRA and DEP+?

5.1. *Evaluation of AntiCRA (RQ1/RQ2)*

For evaluating RQ1 we looked at pure code-reuse attacks; however, at this point such payloads are only rarely found in the wild and are mostly used in academia as proof of concept. The only real-world pure code-reuse exploit we found is a RoP exploit for Adobe Reader. Since neither the exploit's source code nor an infected file is publicly available, our conclusion is based on an analysis by Li and Szor (2013). Analyzing the exploit's source code reveals that the address `0x6acc1049` is repeated 9344 times; the instruction at that address is a simple `ret`. This equals to over 9000 indirect branches in a row, which would, of course, be detected by AntiCRA.

The likely reason for why pure RoP and JoP payloads still seem to be rare in practice is that two-staged payloads (which aim to disable DEP through RoP/JoP) are simpler to construct and are sufficient in many cases. Such payloads can be mitigated by DEP+; but nevertheless, we were interested in evaluating RQ2, i.e., to what extent AntiCRA alone, without DEP+, can be used to mitigate such attacks as well.

We analyzed 11 real-world exploits in total. To operate on an unbiased test set, we analyzed the 10 most recent exploits from http://www.toexploit.com/[1] which claim to bypass ASLR and also added the previously mentioned pure RoP exploit. Fig. 1 and Table 1 show the results of our analysis, i.e., the number of consecutive indirect branches and the average basic block length for each exploit and also for legitimate programs. As the numbers indicate, legitimate programs rarely have more than 15 consecutive indirect branches and their average basic block length is higher than that of exploits. This confirms that our generalized threshholds, which work for a wide variety of programs, are well-suited to detect attacks.

AntiCRA detects 10 out of the 11 exploits in our sample set. In five cases this is due to the number of indirect branches in a row. Three exploits are detected because they use very short gadgets, which mostly only execute one instruction and then transfer program execution to the next gadget. Two exploits trigger both mechanisms, since they use more than 35 indirect branches in a row and also very short gadgets.

One exploit cannot be detected by AntiCRA. This is because it requires only 13 gadgets to prepare the stack for calling `VirtualProtect`. This is not enough to trigger the indirect-branch check. The average length of the basic blocks is 2.2, which would trigger an alarm. However, as explained in Section 3, we only trigger inspections after a total of 15 indirect branches in a row.

It is important to point out that the two-staged exploit AntiCRA misses (E11) is detected by DEP+. AntiCRA is primarily designed to catch pure RoP and JoP attacks, not necessarily the two-staged attacks like the ones examined in the evaluation. It is also important to keep in mind that the thresholds can and should be adjusted for each program and that this section evaluates how well our generalized thresholds work. Despite this, it still detects 10 out of 11 exploits. Because of these results and our analysis of the pure RoP exploit for Adobe

[1] Unfortunately, the website is not available anymore, but the exploits can be requested from the authors of this paper or found online using any search engine and the full name of the exploit, as shown in Table 1.

Reader, we are very confident that exploits which rely solely on RoP or JoP can be detected by AntiCRA.

5.2. *Evaluation of DEP+ (RQ3)*

To test DEP+, we wrote a small vulnerable application, which uses an unbounded `strcpy` and was compiled with the `NX_COMPAT`, and a simple exploit. Since all code injection attacks store the injected code inside a buffer which, by definition, cannot be in an image, the program that contains the vulnerability is of little consequence. The only differences between our vulnerable application and a real application are mitigation techniques which might be in place, but which are irrelevant to us, since we assume an attacker is able to bypass them, and how program flow is transferred to the injected code, which is irrelevant for our evaluation as well. Ultimately, all code injection attacks end up calling their injected code, and this is where DEP + detects them. Therefore, evaluating DEP+ with this self-written program poses no real threat to the validity of this experiment. As expected, DEP+ correctly detects that the target address of the `ret` instruction at the end of our vulnerable function is not in an image, before the instruction is actually executed. Therefore, it can terminate the program and mitigate an attack, which would have led to arbitrary code execution. As for the real world exploits, DEP+ detects each one except for the pure RoP exploit for Adobe Reader, as all the others eventually do execute code from memory outside of images.

5.3. *Performance (RQ4)*

We evaluated the performance of ROPocop using the C and C++ benchmarks in the SPEC CPU2006 benchmark suite. Note that those are really worst-case benchmarks that exercise the dynamic analysis heavily. Any interactive or network-based application would show a significantly lower overhead. We measured five different runtimes for each benchmark:

- The native runtime, i.e., without PIN.
- The runtime with PIN attached, but without instrumentation, to get the basic overhead PIN introduces.
- The runtime with AntiCRA.
- The runtime with DEP+.
- The runtime with AntiCRA and DEP+.

Benchmarks were run on Windows 7 SP1 with an Intel Core 2 Duo T9400 clocked at 2.53 GHz and 4 GB RAM using the reference workload.

Fig. 2 summarizes the results of our performance benchmarks. Running a program under PIN but without any instrumentation introduces an average overhead[2] of 1.36×, i.e., programs take, on average, 36% more time to finish, ranging from 1.002× (470.lbm) to 2.24× (464.h264ref). Programs protected by AntiCRA run, on average, with a total overhead 2.2×. With DEP+ enabled as well, ROPocop introduces an average overhead of 2.39×, which is comparable to similar tools such as ROPdefender (Davi et al., 2011), which gives weaker guarantees. Compared to CFI approaches (Pappas et al., 2013; Zhang et al., 2013; Zhang and Sekar, 2013) ROPocop has a considerably higher overhead; however, it monitors a process throughout its whole lifetime and not just at potentially dangerous points. Thus, it can determine more accurate if a RoP attack is being carried out.

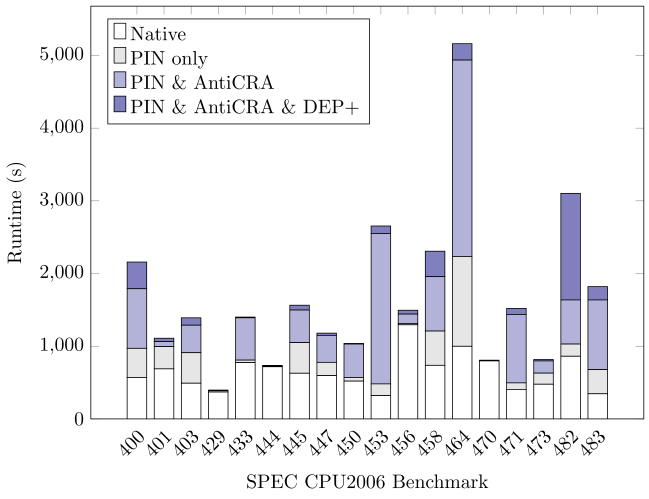

Fig. 2 – Performance of ROPocop. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

While overheads on the order of two-fold might sound unacceptable, those overheads should really only be expected in worst-case situations. Thus, while performance benchmarks such as SPEC CPU2006 are advantageous in producing reproducable results, the results that they do produce do not reflect reality very much. What ultimately counts is the performance on real-world applications. Their performance can, however, often hardly be measured systematically, which is why we only report qualitative results on some of the applications in our sample set. As a general observation we can say that in all cases the GUI had some slight input lag < 1 second when opening a menu for the first time; however, afterwards they opened in an instant. File transfers with Filezilla were no slower than without our tool. VLC plays h.264 encoded HD videos without any jitter. Adobe Reader renders pages without any noticeable lag. Typing in Microsoft Word has no input lag. We want to emphasize that ROPocop is not intended to be used with all applications at all times. Instead, our recommended usage is to enable it only for either very critical systems, or for an application which has a vulnerability that is being actively exploited and no vendor patch has been released yet. Under such circumstances the overhead is, in our opinion, acceptable.

6. Related work

TRUSS (Sinnadurai et al., 2008) and **ROPdefender** (Davi et al., 2011) store copies of return addresses using a runtime shadow stack. When a function is called, a copy of the pushed return

[2] Average overheads were computed using the geometric mean, which is considered best practice for reporting normalized values such as percentages of overhead (Fleming and Wallace, 1986).

address is stored on the shadow stack. Upon returning from a procedure, the return address on the stack is compared to the one on the shadow stack. TRUSS is implemented using DynamoRIO, ROPdefender using PIN. DynamoRIO is a runtime instrumentation tool which works similarly to PIN. TRUSS and ROPdefender rely on an attacker overwriting the return address on the stack, which is not strictly required. Therefore, they can miss some classes of attacks like JoP. Furthermore, they assume that function calls are always made through call and exited via ret. ROPdefender can handle exceptions, but neither can handle hand-crafted assembly code, which does not necessarily follow these conventions. The overhead of both tools is similar to ours.

EMET (Microsoft, 2015) includes, among others, several RoP detection mechanisms, e.g., caller checks, which make sure that critical funtions are invoked via call and not ret, or a routine that detects stack pivoting. However, there is no guarantee that EMET is compatible with the program that should be protected. Furthermore, previous versions of EMET have been bypassed rather quickly after their release.[3,4]

kBouncer (Pappas et al., 2013) makes use of the last branch record (LBR) feature some modern CPUs have. kBouncer assumes that at some point shellcode has to invoke a system call. When this occurs, the LBR repository is checked for distinctive properties of RoP-like behavior, e.g. consecutive indirect jumps and short basic blocks. The tool has an average overhead of only 1%; however, the implementation for Windows 7 is not fully functional, since Windows 7 does not allow to intercept system calls which is a requirement of kBouncer. Furthermore, it cannot be deployed on systems whose CPU does not have LBR.

ROPecker (Cheng et al., 2014) also uses the LBR feature of some modern CPUs. Like ROPocop it checks for consecutive short indirect branches and raises an alert when a certain threshold is undercut. To increase performance the detection heuristic is only invoked if the branch target is outside the so-called "sliding window" (a collection of pages, usually 2 or 4, i.e. 8 or 16 kB). Due to these two circumstances, ROPecker has a very low overhead of only 2.6% for the SPEC CPU2006 benchmark suite. It does, however, miss ROP gadgets which are within the sliding window and requires a CPU which supports the LBR feature.

BinArmor (Slowinska et al., 2012) completely prevents buffer overflows, and therefore stops code redirection and non-control data attacks if they require a buffer overflow. It extracts information about buffers that need protection, either from the debugging symbols or, in the case of stripped binaries, using **Howard** (Slowinska et al., 2011). In a second step, it discovers accesses to those buffers and rewrites the binary to ensure they stay within the bounds of the buffer. **BinArmor**'s overhead lies typically between 2 and 3×.

Control Flow Integrity (Abadi et al., 2009) uses static analysis of a binary to create a control-flow graph and rewrites the binary to enforce it does not deviate from the pre-computed paths. The implementation is based on Vulcan, a commercial dynamic instrumentation tool for x86 binaries. The average overhead is about 16%.

CCFIR (Zhang et al., 2013) enforces control-flow integrity by ensuring that targets of indirect jumps are legal. Valid targets are identified ahead of time by statically analysing a given binary. For their analysis to work properly, they require the binary to use ASLR and DEP. CCFIR has a runtime overhead of about 4%.

Göktaş et al. (2014) have recently shown that the above mentioned CFI approaches can be bypassed. The inherent problems of these approaches is that there are too few checks and/or they are to coarse-grained, allowing attackers to access too many gadgets. They are further limited by the number of slots in the LBR, which is at most 16. To improve the security of approaches which attempt to detect RoP exploits by measuring similar properties as we do, they suggest making the thresholds dynamic.

7. Conclusion

In this work we have presented ROPocop, a novel tool for the automated dynamic recognition of buffer-overflow attacks. ROPocop is designed to recognize different classes of code-reuse attacks based on two novel techniques AntiCRA and DEP+. AntiCRA is a configurable heuristic based on the number of indirect branches executed in a row as well as on the average basic block length of executed code. In our experiments using default thresholds which work for a variety of programs, AntiCRA detects 10 out of 11 of the latest real-world code-reuse exploits and yields no false alarms on SPEC CPU2006 and all tested real-world applications, a total of 35 programs. DEP+ executes a non-executable stack through binary instrumentation and can thus be used to detect exploits based on two-staged payloads that use a code-reuse attack to disable DEP using the Windows API. DEP+ successfully detects all two-staged payloads we examined, again with no false alarms. By combining both techniques, ROPocop thus successfully detects all tested exploits, without false warnings, showing an average performance overhead of 2.4× for SPEC CPU2006 and real-world applications showing only an initially noticeable input lag and no stutter. ROPocop runs in user mode, requiring no access to source code, nor debug symbols or changes to the operating system. It supports multi-threaded applications. Due to its heuristic nature, ROPocop cannot give an absolute security guarantee. However, the parameters the heuristic is based on should make it very hard to circumvent the approach in practice. ROPocop is thus raising the bar significantly, without any added cost compared to previous related approaches.

Acknowledgements

This work was supported by the BMBF within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, and by the DFG Collaborative Research Center CROSSING.

[3] <https://www.offensive-security.com/vulndev/disarming-and-bypassing-emet-5-1/>.

[4] <http://casual-scrutiny.blogspot.de/2015/03/defeating-emet-52.html>.

REFERENCES

Abadi M, Budiu M, Erlingsson U, Ligatti J. Control-flow integrity principles, implementations, and applications. ACM Trans Inform Syst Secur 2009;13(1):4:1–40. doi:10.1145/1609956.1609960 <http://doi.acm.org/10.1145/1609956.1609960>.

Andersen S, Abella V. Changes to functionality in Windows XP service pack 2 — part 3: memory protection technologies, <http://technet.microsoft.com/en-us/library/bb457155.aspx>; 2004 [accessed 19.07.13].

Bletsch T, Jiang X, Freeh VW, Liang Z. Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS '11. ACM; 2011. p. 30–40. doi:10.1145/1966913.1966919 <http://doi.acm.org/10.1145/1966913.1966919>.

Bubinas C. Buffer overflows are the top software security vulnerability of the past 25 years, <http://www.klocwork.com/blog/software-security/buffer-overflows-are-the-top-software-security-vulnerability-of-the-past-25-years>; 2013 [accessed 19.07.13].

Butler J. Anonymous, Bypassing 3rd party windows buffer overflow protection. Phrack 2004;11.

Checkoway S, Davi L, Dmitrienko A, Sadeghi A-R, Shacham H, Winandy M. Return-oriented programming without returns, CCS '10. ACM; 2010. p. 559–72. doi:10.1145/1866307.1866370 <http://doi.acm.org/10.1145/1866307.1866370>.

Chen S, Xu J, Sezer EC, Gauriar P, Iyer RK. Non-control-data attacks are realistic threats. In: Proceedings of the 14th Conference on USENIX Security Symposium — Volume 14, SSYM'05. USENIX Association; 2005. p. 12 <http://dl.acm.org/citation.cfm?id=1251398.1251410>.

Cheng Y, Zhou Z, Yu M, Ding X, Deng RH. Ropecker: a generic and practical approach for defending against ROP attacks. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23–26, 2014; 2014 <http://www.internetsociety.org/doc/ropecker-generic-and-practical-approach-defending-against-rop-attacks>.

Davi L, Sadeghi A-R, Winandy M. Ropdefender: a detection tool to defend against return-oriented programming attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS '11. ACM; 2011. p. 40–51. doi:10.1145/1966913.1966920 <http://doi.acm.org/10.1145/1966913.1966920>.

Davi L, Sadeghi A-R, Lehmann D, Monrose F. Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In: Proceedings of the 23rd USENIX Conference on Security, SEC'14. USENIX Association; 2014. p. 401–16 <http://dl.acm.org/citation.cfm?id=2671225.2671251>.

Durden T. Bypassing pax aslr protection. Phrack 2002;11.

Fleming PJ, Wallace JJ. How not to lie with statistics: the correct way to summarize benchmark results. Commun ACM 1986;29(3):218–21. doi:10.1145/5666.5673. <http://doi.acm.org/10.1145/5666.5673>.

Göktaş E, Athanasopoulos E, Polychronakis M, Bos H, Portokalidis G. Size does matter: why using gadget-chain length to prevent code-reuse attacks is hard. In: Proceedings of the 23rd USENIX Conference on Security Symposium, SEC '14. USENIX Association; 2014. p. 417–32 <http://dl.acm.org/citation.cfm?id=2671225.2671252>.

Gunn A. Pwn2own 2014: a recap, <http://www.pwn2own.com/2014/03/pwn2own-2014-recap/>; 2014 [accessed 05.04.13].

Howard M, Miller M, Lambert J, Thomlinson M. Windows isv software security defenses, <http://msdn.microsoft.com/en-us/library/bb430720.aspx>; 2010 [accessed 16.07.13].

Hund R, Willems C, Holz T. Practical timing side channel attacks against kernel space aslr. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP '13. IEEE Computer Society; 2013. p. 191–205. doi:10.1109/SP.2013.23 <http://dx.doi.org/10.1109/SP.2013.23>.

Li X, Szor P. Emerging stack pivoting exploits bypass common security, <http://blogs.mcafee.com/mcafee-labs/emerging-stack-pivoting-exploits-bypass-common-security>; 2013 [accessed 16.07.13].

Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, et al. Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation, PLDI '05. ACM; 2005. p. 190–200. doi:10.1145/1065010.1065034 <http://doi.acm.org/10.1145/1065010.1065034>.

Microsoft. Data execution prevention, <http://technet.microsoft.com/en-us/library/cc738483(v=ws.10).aspx>; [accessed 16.07.13].

Microsoft. Enhanced mitigation experience toolkit 5.5 beta user guide, <http://download.microsoft.com/download/4/3/3/43364390-96B1-4820-9BAD-4A71F9A3221A/EMET%20User%20Guide.pdf>; 2015 [accessed 20.10.15].

Min J-W, Jung S-M, Lee D-Y, Chung T-M. Jump oriented programming on windows platform (on the x86). In: Murgante B, Gervasi O, Misra S, Nedjah N, Rocha AMAC, Taniar D, et al., editors. ICCSA (3), Vol. 7335 of Lecture Notes in Computer Science. Springer; 2012. p. 376–90 <http://dblp.uni-trier.de/db/conf/iccsa/iccsa2012-3.html#MinJLC12>.

One A. Smashing the stack for fun and profit. Phrack 1996;7.

Pappas V, Polychronakis M, Keromytis AD. Transparent rop exploit mitigation using indirect branch tracing. In: Proceedings of the 22nd USENIX Conference on Security, SEC '13. USENIX Association; 2013. p. 447–62 <http://dl.acm.org/citation.cfm?id=2534766.2534805>.

Roemer R, Buchanan E, Shacham H, Savage S. Return-oriented programming: systems, languages, and applications. ACM Trans Inform Syst Secur 2012;15(1):2:1–34. doi:10.1145/2133375.2133377. <http://doi.acm.org/10.1145/2133375.2133377>.

Shacham H. 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, CCS '07. ACM; 2007. p. 552–61 doi:10.1145/1315245.1315313 <http://doi.acm.org/10.1145/1315245.1315313>.

Shacham H, Page M, Pfaff B, Goh E-J, Modadugu N, Boneh D. On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS '04. ACM; 2004. p. 298–307. doi:10.1145/1030083.1030124 <http://doi.acm.org/10.1145/1030083.1030124>.

Sinnadurai S, Zhao Q, Wong W-F. Transparent runtime shadow stack: protection against malicious return address modifications, 2008.

Slowinska A, Stancescu T, Bos H. Howard: a dynamic excavator for reverse engineering data structures. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6 February–9 February 2011; 2011 <http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_1.pdf>.

Slowinska A, Stancescu T, Bos H. Body armor for binaries: preventing buffer overflows without recompilation. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC '12. USENIX Association; 2012. p. 11 <http://dl.acm.org/citation.cfm?id=2342821.2342832>.

Zhang C, Wei T, Chen Z, Duan L, Szekeres L, McCamant S, et al. Practical control flow integrity and randomization for binary executables. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP '13. IEEE Computer Society; 2013. p. 559–73. doi:10.1109/SP.2013.44 <http://dx.doi.org/10.1109/SP.2013.44>.

Zhang M, Sekar R. Control flow integrity for cots binaries. In: Proceedings of the 22nd USENIX Conference on Security, SEC '13. USENIX Association; 2013. p. 337–52 <http://dl.acm.org/citation.cfm?id=2534766.2534796>.