eavesROP: Listening for ROP Payloads in Data Streams

Christopher Jämthagen, Linus Karlsson, Paul Stankovski, and Martin Hell

Dept. of Electrical and Information Technology, Lund University,
P.O. Box 118, 221 00 Lund, Sweden
{christopher.jamthagen,linus.karlsson,paul.stankovski,martin.hell}@eit.lth.se

Abstract. We consider the problem of detecting exploits based on return-oriented programming. In contrast to previous works we investigate to which extent we can detect ROP payloads by only analysing streaming data, i.e., we do not assume any modifications to the target machine, its kernel or its libraries. Neither do we attempt to execute any potentially malicious code in order to determine if it is an attack. While such a scenario has its limitations, we show that using a layered approach with a filtering mechanism together with the Fast Fourier Transform, it is possible to detect ROP payloads even in the presence of noise and assuming that the target system employs ASLR. Our approach, denoted eavesROP, thus provides a very lightweight and easily deployable mitigation against certain ROP attacks. It also provides the added merit of detecting the presence of a brute-force attack on ASLR since library base addresses are not assumed to be known by eavesROP.

Keywords: Return-Oriented Programming, ROP, Pattern Matching, ASLR

1 Introduction

Buffer overruns [1] have for a long time been a common source of software vulnerabilities. The buffer overrun vulnerability may be exploited to perform a code injection attack, where the goal is to inject arbitrary data and replacing the return address with the address of the injected data. There are several well-known and widely used mitigations against this approach. Since the injected code should not be executable, but rather considered as data, the memory pages corresponding to this data can be marked as non-executable. Data Execution Prevention (DEP) [14] and the $W \oplus X$ security feature [22] are approaches implementing this idea. While this will prevent the classic code injection attacks, it will not prevent code reuse attacks. In these attacks, the adversary will not inject the code to be executed, but will instead direct the program flow to code that is already loaded by the process, typically a shared library. One example is the return-to-libc attack [4] in which the attack points to existing code in the libc library.

A more advanced code reuse attack is Return-Oriented Programming (ROP), in which the attacker identifies small pieces of usable code segments, called *gadgets*, and chains them together using a `ret` instruction. A `ret` instruction will pop an address from the stack and continue execution at that address.

One available countermeasure against code injection, which can also be applied to prevent code reuse attacks, is Address Space Layout Randomization (ASLR) [22]. ASLR will randomize the base address of the program's text, stack, and heap segments and the adversary will not know where the gadgets will be located. However, as described in Section 2.2, ASLR can sometimes be bypassed.

There have been several proposed defenses against ROP attacks, all taking slightly different approaches and using different assumptions. A typical mitigation is to identify some specific features in the attack that distinguishes it from benign code execution and then build a mitigation technique based on those distinguishing features [7,8,10,12,21,23]. Another approach is to rewrite libraries or targeted code such that it is not usable in an attack [17,19] or to randomize addresses which are needed by an attacker [13,15,20,32].

Instead of detecting the attacks on the target systems, another goal may be to detect ROP attacks in data. In [23] data was scanned and possible exploits were speculatively executed in order to determine if they were exploits. This requires a snapshot of the virtual memory of the process that is protected. In [29] the authors consider a detection approach where documents are analyzed to find ROP attacks. Documents are collected and sent to a separate virtual machine, where they are opened in their native application, and the memory is then analyzed for ROP payloads.

In this paper we present eavesROP, which is a more lightweight approach where no execution takes place. We try to identify ROP payloads by looking at network traffic only, i.e., we do not make any modifications to machines, programs, libraries or operating systems, nor do we try to execute any of the received data. We do not even require any kind of access to the machines. Scenarios could be an implementation in a gateway to a corporate network, ROP payload detection in switches or at an ISP before data is forwarded to the end user. The question that we try to answer is: How much information can we deduce by just looking at the data? We target ROP exploits where gadget addresses can be explicitly found in the data sent to the application. We assume that ASLR is enforced by the operating system, and that the attacker has somehow acquired information about the location of libraries. Of course, our detection mechanism has no such information. We show how to filter out possible ROP payloads and how to determine if the candidate payload is a ROP attack or not. Even with just a moderate number of gadgets, we can detect the payload efficiently. This is true even if there is a large amount of noise present.

We have tested eavesROP using available exploits and it is able to detect exploits with no false negatives.

2 Background

In this section we provide the necessary background on return-oriented programming and address space layout randomization.

2.1 Return-Oriented Programming

Return-oriented programming [26] is an exploitation technique that allows for arbitrary code execution without having to inject code into the vulnerable process. To achieve this, an attacker constructs a payload of addresses, each pointing to a small sequence of instructions reachable and executable by the affected process. These instruction sequences are called gadgets and typically consist of very few instructions, ending with a return instruction (`ret`). This return instruction will pop the next dword from the stack, put it into the instruction pointer register (EIP) and continue execution at the next gadget. Gadgets do not have to be aligned with the intended instructions. Any byte that represents the opcode of a `ret` can potentially be used as a gadget.

Not only `ret` instructions can be used in these types of attacks. It is also possible to use jump-based instructions as in [2,6]. We will not consider these type of attacks in this paper, but note that it would be possible to extend our algorithms to detect these gadgets as well.

2.2 Address Space Layout Randomization (ASLR)

ASLR protects from buffer overflow attacks by randomizing the location of the stack, the heap and the location of all dynamically loaded libraries. The term was coined by the PaX project [22] which also has a well-known implementation.

Following the introduction of ASLR in Windows XP SP2 (2004) and in the Linux Kernel (since version 2.6.12, 2005), writing exploits has become much more difficult.

However, the efficiency of ASLR is limited. First, some small amount of code is not randomized, leaving the possibility to still use gadgets in the code where the location is predictable. Even though this code is rather small, it has been shown that it is possible to find usable gadgets in it [24]. Randomizing the application code is one kind of protection against these attacks [32]. Another aspect of ASLR, as was shown in [27], is the limited entropy in the address space, which makes it possible to brute-force absolute locations.

In addition to brute-forcing the ASLR, it has been shown that information leakage can occur through e.g., buffer and heap overrun bugs [11,31] and other types of vulnerabilites [25,28]. This could give an attacker at least partial information about the location of ASLR-affected code.

The exact means by which an attacker bypasses ASLR, may it be through brute force or information leakage, are independent of our payload detection algorithm.

3 Our Approach

In this section we give a description of the different parts of eavesROP. The idea is based on the observation that many gadgets are typically taken from the same library which results in gadget addresses being located relatively close to each other. Note though, that our approach only require a few gadgets to be located in the same library, other gadgets can be taken from other libraries. A more detailed description of eavesROP can be found in the full version of this paper [16].

3.1 Optional Data Pre-Filter

Certain input data can be expected to exhibit properties that make them look like addresses close to each other in the memory space—thus looking like ROP payloads—even though the data is actually non-malicious. Our goal is to filter out this data before it reaches later steps in the algorithm, to reduce the total computational overhead of our system.

Of special interest are printable ASCII characters, not only because much data is readable text, but also because large portions of adjacent ASCII data may—when combined into 32-bit words—look like adjacent addresses. Filtering is however a trade-off between performance and false negatives. There are ways to make ROP payloads printable [18]. Such a payload would be removed if a filter for printable characters is enabled. This is why the filtering step should be considered optional.

If the pre-filter is enabled, it removes blocks of UTF-8 strings. In our implementation, we define a block as a sequence of five or more adjacent, printable UTF-8 characters. When a matching block is found, the complete block is removed from the input. This leads to potential noise as non-adjacent bytes become adjacent after the data between them is removed. This does, however, only affect a few addresses, which does not cause any problems in practice since our ROP pattern matching is very precise and noise tolerant.

3.2 Cluster Detection

An actual exploit payload will contain several gadget addresses that lie close together with respect to the entire addressable memory space. The purpose of the address cluster detection is to find and isolate the congested parts of the memory space for further processing by generating a binary address vector, P_{obs}, of size L, i.e., the size of the largest targeted library. These vectors indicate addresses found in the data.

We let M denote the maximum size of a ROP payload in 4-byte words that our detection is guaranteed to support. A naïve approach to detect the gadget addresses is to pick M words of data, map them to P_{obs} and match this vector with a known library pattern P_{lib}. Doing this byte by byte in the data would produce the correct maximum matching, but it is a very slow approach. Moreover, all words but one will repeat every 4 bytes. Another problem is that the addresses contained within the data window can be spread out over the entire ASLR address space (N bytes), making P_{obs} very large. We propose to use an algorithm that is much more efficient, and will still always find the correct maximum matching.

Instead of considering M addresses, we pick a data window of size $D = 2M$. Thus, we consider twice as much data as the maximum payload, but in return we consider $M + 1$ possible payloads simultaneously. Doubling the data window size introduces more noise (more data in one window), so a few more data windows will pass the cluster detection stage, but this effect is marginal compared to the significant gain in processing efficiency.

When the data window slides over the next data chunk of size M, we begin by extracting potentially viable addresses. As the offset of a ROP payload in the data buffer is unknown, but we know that each address is four bytes and addresses are aligned inside a payload, we create a list for each offset.

We need to keep track of eight such address lists, four for each of the two M-word data chunks covered by the D-word data window. Separating the lists per data chunk allows for incrementing the data window in steps of M words, while reusing the four lists corresponding to the previous M words.

The four new address lists are sorted using an efficient linear-time sorting algorithm such as bucket sort [9]. Such efficient sorting is possible since all addresses are of the same size. Once sorted, we slide an address window of size L (same size as executable part of the largest library) down the combination of the two lists for each offset. Since each of the two lists is individually sorted, we can easily traverse the sorted combination of the lists with time complexity $\mathcal{O}(D)$.

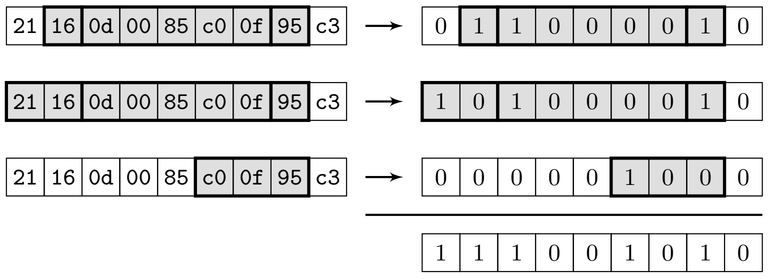
Let T be a threshold value that determines the minimum number of gadgets in an exploit that we want to be able to detect. A small T leads to detection of more exploits, but it also results in more pattern matching, slowing down the detection algorithm. In practice, the lowest value that our algorithm can handle is $T \approx 6$, depending on the instruction size of each gadget (see Section 3.3) and the error probabilities (see Section 3.4).

If we find an address window that contains at least T unique addresses, the binary vector P_{obs} is constructed by entering a '1' in each position corresponding to an address in the address window. To minimize redundant checks, P_{obs} is normalized to always start with a '1'. Then we proceed to perform pattern matching via FFT, as described in Section 3.3.

3.3 Pattern Matching

The vector P_{obs} from the previous step is matched with binary library vectors. The relative distances of the memory addresses in an address window form a very distinct pattern. This pattern is matched with the gadget address patterns of libraries P_{lib}.

Since we allow ASLR, the precise memory location of the library is unknown. This is handled by using the Fast Fourier Transform to compute the maximum matching between P_{obs} and P_{lib}.



21 16 0d 00 85 c0 0f 95 c3
0 1 1 0 0 0 0 1 0
21 16 0d 00 85 c0 0f 95 c3
1 0 1 0 0 0 0 1 0
21 16 0d 00 85 c0 0f 95 c3
0 0 0 0 0 1 0 0 0
1 1 1 0 0 1 0 1 0

Fig. 1. Translation of maximal length gadget sequences to binary pattern.

Identifying Gadgets in a Library In order to find all possible gadgets in a library, the executable part of it is scanned for the opcode of different types of return instructions, namely `0xC2` (`retn imm16`), `0xC3` (`retn`), `0xCA` (`retf imm16`) and `0xCB` (`retf`). For each position of these bytes in the library we search backwards one byte at a time and try to assemble a legal instruction flow ending with the return. We define the *entry zone*, z, as the number of instructions we allow for each gadget, not including the return instruction. This means that we can find many gadgets ending at the same return instruction due to the possibility of instruction overlapping in the x86 architecture.

The starting byte of every possible gadget is used to construct the binary vector P_{lib}. This is the vector that is used for pattern matching with P_{obs}, which is the output of the address cluster detection algorithm.

To understand how the gadget structure in a library is translated into a binary pattern, consider the following sequence of nine bytes (hex):

```
21 16 0d 00 85 c0 0f 95 c3
```

Using an entry zone of size $z = 3$ (at most three instructions), we construct maximal gadget chains by interpreting the bytes preceding the return instruction `c3` as consecutive instructions. There are three possible maximal gadget chains in the above byte sequence, as illustrated in Figure 1.

The top two gadget chains are both of length three. While the top chain begins with a single-byte instruction `16`, the second chain extends this to a two-byte instruction `21 16`. The third chain is of length 1, but it is maximal since it cannot be further extended.

A sequence of bytes belonging to a library is translated into a binary pattern. A '1' in the array represents a gadget and a '0' is used for the other positions.

Pattern Matching via FFT Perfect pattern matching can be performed efficiently using a Fast Fourier Transform (FFT). Pattern matching, here, means that we want to find the maximum weight of the overlap between two patterns that are overlaid. We also want this matching to be perfect, which is to say that all actual gadget addresses that are used in an exploit will be counted. All actual gadget addresses in an exploit will, in the general case, contribute positively to the weight of the maximal pattern match.

Focusing on one library, P_{lib} and P_{obs} are binary vectors of length L. If both patterns are aligned, the maximum matching can be calculated as the dot product between P_{lib} and P_{obs} according to

$$P_{\text{lib}} \cdot P_{\text{obs}} = \sum_{i=0}^{L-1} P_{\text{lib}}[i] P_{\text{obs}}[i].$$

However, we have no way of knowing if the alignment is correct, so we rather need to try all alignments to see which one produces the highest fit. That is, we need to calculate the dot products for all possible shifts of the two patterns. This can be accomplished by using the Fast Fourier Transform (FFT). The FFT computes the circular discrete convolution c of two vectors a and b of length L,

$$c\,[t] = (a * b_L)\,[t] = \sum_{i=0}^{L-1} a\,[i]\,b\,[(t-i) \mod L]\,. \tag{1}$$

For this to be applicable to our situation, we need to adjust two things. First of all, we need to reverse one of the vectors, say P_{lib}. Secondly, since indices in Eq. (1) are taken modulo L, we need to pad both P_{lib} and P_{obs} with zeros to double length. Without this zero padding, the tails and fronts of the two vectors will contribute to the maximum matching in an undesirable way, effectively bringing more noise into our result.

The FFT approach (see [3]) has time complexity $O(L \lg L)$, compared to $O(L^2)$ for the naïve approach. Letting $\mathcal{F}$ denote the FFT version of the Discrete Fourier Transform (DFT), we may compute c as

$$c = \mathcal{F}^{-1}\left(\mathcal{F}\,(a) \odot \mathcal{F}\,(b)\right),$$

where $\odot$ denotes componentwise multiplication.

We let a and b be the vectors P_{lib} and P_{obs} respectively after the zero padding as described above. The weight of the maximum matching is given as the maximum component of c,

$$c_{\max} = \max_i c\,[i]\,. \tag{2}$$

Note that P_{lib} is known beforehand, so we can precompute $\mathcal{F}\,(a)$ for efficiency.

3.4 Statistical Test

The maximum overlap is given by the maximum value of the inverse Fourier transform as given in Eq. (2). In order to find an expression for the number of overlaps we make the following approximations.

- Locations corresponding to gadgets in P_{lib} are uniformly distributed.
- The entries in the convolution vector c are approximated as independent events, all with the same probability.

Using these approximations, the number of overlaps between P_{lib} and P_{obs} is binomially distributed, $X^{(w)} \sim \text{Bin}(w, \frac{G}{L})$, where w and G denote the Hamming weights of P_{obs} and P_{lib}, respectively. Note that G should here be understood as the number of gadgets in a library for a given entry zone, and w is the number of addresses in an address window. Thus, the probability that there are s overlaps is given by

$$\Pr(X^{(w)} = s) = \binom{w}{s} \left(\frac{G}{L}\right)^{s} \left(1 - \frac{G}{L}\right)^{w-s}. \tag{3}$$

Since P_{lib} and P_{obs} are convolved, each convolution consists of L such binomially distributed samples. In order to find the probability distribution for the maximum value of the convolution array, we write the probability that any single value is at most s as

$$\Pr(X^{(w)} \leq s) = \sum_{t=0}^{s} \binom{w}{t} \left(\frac{G}{L}\right)^{t} \left(1 - \frac{G}{L}\right)^{w-t}.$$

The probability that all values are at most s is then, using the second approximation above, $\Pr(X^{(w)} \leq s)^L$. From this it follows that the probability distribution for the maximum value of the convolution vector $c_{\max}^{(w)}$ is given by

$$f_{c_{\max}^{(w)}}(s) = \Pr(c_{\max}^{(w)} = s) = \Pr(X^{(w)} \leq s)^L - \Pr(X^{(w)} \leq s-1)^L \tag{4}$$

with cumulative distribution function

$$F_{c_{\max}^{(w)}}(s) = \Pr(c_{\max}^{(w)} \leq s) = \Pr(X^{(w)} \leq s)^L.$$

A threshold value for $c_{\max}^{(w)}$ is chosen, denoted $\hat{c}_{\max}$. If $c_{\max}^{(w)} \geq \hat{c}_{\max}$ the payload is considered a ROP. Associated with this decision are false positives and false negatives. The false positive rate, denoted α, is defined as the probability that non-malicious data is considered malicious (i.e., a ROP payload) while the false negative rate, denoted β, is the probability that a malicious payload is mistaken for non-malicious data. To write expressions for α and β, let the Hamming weight w of P_{obs} be written as $w = w_G + w_N$, where w_G is the number of ROP gadgets and w_N is the number of noise addresses. The distribution of $c_{\max}^{(w)}$ for non-malicious data is given by Eq. (4). The value of $c_{\max}^{(w)}$ for a ROP payload is given by $c_{\max}^{(w)} = w_G + X^{(w_N)}$, where $X^{(w_N)}$ is distributed according to Eq. (3). Now, we can write the two error probabilities as

$$\begin{aligned} \alpha =& \Pr(c_{\max}^{(w)} \geq \hat{c}_{\max}) = 1 - \Pr(c_{\max}^{(w)} \leq \hat{c}_{\max} - 1) \\ =& 1 - \Pr(X^{(w)} \leq \hat{c}_{\max} - 1)^L \\ \beta =& \Pr(X^{(w_N)} < \hat{c}_{\max} - w_G) = \Pr(X^{(w_N)} \leq \hat{c}_{\max} - w_G - 1) \end{aligned} \tag{5}$$

The false positives rate α is only for one library. If we want to test the payload against a set of ℓ libraries, the total false positive rate α_ℓ is given by $\alpha_\ell = 1 - (1-\alpha)^\ell$.

By choosing $\alpha = 0.0001$ we allow $\ell = 100$ libraries to be supported, still keeping the total false positive rate α_ℓ below 0.01. (We assume here that all libraries are of approximately equal size.) We let the false negative rate $\beta = 0.01$ since this is not affected by multiple libraries (the payload will only match one library). Using these values for α and β allows us to compute the threshold $\hat{c}_{\max}$ and the minimum number of gadgets w_G that are required for successful detection. Table 1 gives these numbers for $z = 3$ and some different choices of w. Note that for all values w such that $\hat{c}_{\max} = w_G$, we have $\beta = 0$. Thus we will only obtain false negatives for very large noise values. For $z = 1$ we will get $\hat{c}_{\max} = w_G = 6$, which is the lowest possible threshold T for eavesROP.

Table 1. Threshold $\hat{c}_{\max}$ and minimum number w_G of gadgets needed for ROP payload detection in an address window of weight w. Error rates $\alpha \leq 0.0001$ and $\beta \leq 0.01$. The example library used is libc 2.18 of size $L = 1224144$.

	entity	values								
weight of P_{obs}	w	7	10	15	20	25	30	50	100	200
threshold	$\hat{c}_{\max}$	7	8	9	10	11	12	15	20	27
min num gadgets	w_G	7	8	9	10	11	12	15	20	26

The standard deviation of Eq. (4) turns out to be very small, with almost all probability mass concentrated to only a few values for s. This makes the detection algorithm efficient, allowing us to choose small error rates while still requiring few gadgets to succeed, even in the presence of a large amount of noise.

It can be noted that the required number of gadgets w_G is very close to the threshold $\hat{c}_{\max}$. Thus, as a rule of thumb, the number of gadgets required for successful detection is approximately equal to the threshold $w_G \approx \hat{c}_{\max}$.

The false positive rate has been simulated using the data from Table 2. The simulations indicate that the actual false positive rate is slightly larger than that given by Eq. (5). This is not surprising, since the theoretical model assumes that gadget addresses are uniformly distributed. Due to data redundancy and the proximity coupling between gadgets and return instructions, a slightly larger $c_{\max}^{(w)}$ is expected. Still, according to our simulations, increasing the threshold value by 2 will remove virtually all false positives. This shows that the theoretical model is adequate.

3.5 Performance

The performance of eavesROP depends on the parameters used in the various stages of the system. All simulations have been performed on an Intel Core i7 4770 @ 3.4 GHz with 16 GB of RAM.

A more aggressive filtering in each step will reduce the amount of data sent to the next stage, which will increase the overall performance. Simulations have been performed on various types of data. The data pre-filter has a throughput of approximately 35 MiB/s for compressed or random data, and approximately 50 MiB/s for web data (HTML, JPEG, ...). The input/output size ratio varies between 0.965 for random or compressed data, to 0.068 for web data.

After the optional data pre-filter—which may have reduced the total amount of data—the data is passed to the cluster detection step. This step has a throughput of around 10 MiB/s. The output of the cluster detection step is multiple matched windows, i.e. multiple P_{obs}. Table 2 shows how many P_{obs} vectors that are passed to the pattern matching layer, for some different parameters.

Table 2. Number of matching address windows per GiB of input data, for a data window of size D, and with at least T addresses within distance $L = 1224144$, for different types of data. L is here the size of libc 2.18.

	$D = 50$			$D = 200$			$D = 1000$		
type of data	$T = 6$	8	10	$T = 6$	8	10	$T = 6$	8	10
random	0	0	0	12	0	0	24749	53	0
web (HTML, JPG,...)	1795	689	590	5589	1878	1208	40795	8007	3292
mp3	42	8	2	631	106	10	162014	8472	1023
pdf	4068	248	61	34718	5266	1316	1011850	176992	45289
mkv (H.264/MPEG-4)	354	2	0	513	81	66	35545	841	125

Each P_{obs} outputted from the cluster detection stage will be passed to the pattern matching step. Each pattern matching sequence takes roughly 1 second using FFT implemented in software. If necessary, this step could be accelerated using a hardware FFT implementation.

All parts of eavesROP have been implemented and tested using real-world exploits. We are able to detect all exploits of at least 6 gadgets, using a threshold value $\hat{c}_{\max} = 6$, for example [5] and [30]. More simulation results can be found in the full version of the paper [16].

4 Conclusions

We have investigated to which extent it is possible to detect a ROP payload by only analysing data, and assuming that ASLR is used on the target system. If we have the set of libraries and binaries that can be used to find gadgets, we show that it is possible to detect a ROP payload even in the presence of noise and by applying suitable data filters and the Fast Fourier Transform the detection has acceptable performance. Naturally, encrypted traffic, obfuscated ROP payloads and locally generated exploits are out of scope for our detection approach. The exact performance will depend on the type of data and the number of gadgets that are required for an exploit to be detected depends on the maximum allowed size for the payload and the amount of noise. Furthermore performance may be optimized with greater pre-filtering and dedicated hardware for FFT calculations.

References

1. Aleph One: Smashing the stack for fun and profit, phrack, 49 (1996)
2. Bletsch, T., Jiang, X., Freeh, V.W., 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. pp. 30–40. ASIACCS '11, ACM, New York, NY, USA (2011)
3. Bracewell, R.: The Fourier Transform and its Applications. McGraw-Hill Series in Electrical and Computer Engineering, McGraw-Hill Science/Engineering/Math; 3 edition (June 1999)
4. c0ntex: Bypassing non-executable-stack during exploitation using return-to-libc. Available at: `http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf`
5. Cantoni, L.: BigAnt Server 2.52 SP5 - SEH Stack Overflow ROP-based exploit (ASLR + DEP bypass). Available at: `http://www.exploit-db.com/exploits/22466/`
6. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. pp. 559–572. CCS '10, ACM, New York, NY, USA (2010)
7. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: Drop: Detecting return-oriented programming malicious code. In: Information Systems Security, Lecture Notes in Computer Science, vol. 5905. Springer Berlin Heidelberg (2009)
8. Cheng, Y., Zhou, Z., Miao, Y., Ding, X., Deng, R.: ROPecker: A generic and practical approach for defending against ROP attack. In: NDSS. Research Collection School Of Information Systems (2014)
9. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, Third Edition. MIT Press (2009)
10. Davi, L., Sadeghi, A., 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 (2011)
11. Durden, T.: Bypassing PaX ASLR protection, phrack, 59 (2002)
12. Fratric, I.: Ropguard: Runtime prevention of return-oriented programming attacks (2012)
13. Gupta, A., Kerr, S., Kirkpatrick, M., Bertino, E.: Marlin: Making it harder to fish for gadgets. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. CCS '12, ACM (2012)
14. Hensing, R.: Understanding DEP as a mitigation technology. Available at: `http://blogs.technet.com/b/srd/archive/2009/06/12/understanding-dep-as-amitigation-technology-part-1.aspx` (2009)
15. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.: Ilr: Where'd my gadgets go? In: Security and Privacy (SP), 2012 IEEE Symposium on (2012)
16. Jämthagen, C., Karlsson, L., Stankovski, P., Hell, M.: eavesROP: Listening for ROP payloads in data streams (full version). Available at: `http://lup.lub.lu.se/record/4586662` (2014)

17. Li, J., Wang, Z., Jiang, X., Grace, M., Bahram, S.: Defeating return-oriented rootkits with "return-less" kernels. In: Proceedings of the 5th European Conference on Computer Systems. EuroSys '10, ACM (2010)
18. Lu, K., Zou, D., Wen, W., Gao, D.: Packed, printable, and polymorphic return-oriented programming. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) Recent Advances in Intrusion Detection, Lecture Notes in Computer Science, vol. 6961, pp. 101–120. Springer Berlin Heidelberg (2011)
19. Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-free: Defeating return-oriented programming through gadget-less binaries. In: Proceedings of the 26th Annual Computer Security Applications Conference. pp. 49–58. ACSAC '10, ACM (2010)
20. Pappas, V., Polychronakis, M., Keromytis, A.: Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: IEEE Symposium on Security and Privacy. IEEE Computer Society (2012)
21. Pappas, V., Polychronakis, M., Keromytis, A.: Transparent ROP exploit mitigation using indirect branch tracing. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13). USENIX (2013)
22. PaX Team: Address space layout randomization. Available at: `http://pax.grsecurity.net/docs/aslr.txt` (2003)
23. Polychronakis, M., Keromytis, A.: ROP payload detection using speculative code execution. In: Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software. MALWARE '11, IEEE Computer Society (2011)
24. Schwartz, E., Avgerinos, T., Brumley, D.: Q: Exploit hardening made easy. In: Proceedings of USENIX Security 2011 (2011)
25. Serna, F.J.: CVE-2012-0769, the case of the perfect info leak. Available at: `http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf` (2009)
26. 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. pp. 552–561. CCS '07, ACM (2007)
27. Shacham, H., Page, M., Pfaff, N., Goh, E., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security. pp. 298–307. CCS '04, ACM (2004)
28. Snow, K., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Security and Privacy (SP), 2013 IEEE Symposium on. pp. 574–588 (May 2013)
29. Stancill, B., Snow, K.Z., Otterness, N., Monrose, F., Davi, L., Sadeghi, A.R.: Check my profile: Leveraging static analysis for fast and accurate detection of ROP gadgets. In: Stolfo, S., Stavrou, A., Wright, C. (eds.) Research in Attacks, Intrusions, and Defenses, Lecture Notes in Computer Science, vol. 8145, pp. 62–81. Springer Berlin Heidelberg (2013)
30. Sud0: Audio converter 8.1 0day stack buffer overflow PoC exploit ROP/WPM. Available at: `http://www.exploit-db.com/exploits/13763/`
31. Vreugdenhil, P.: Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. Available at: `http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf` (2010)
32. Wartell, R., Mohan, V., Hamlen, K., Lin, Z.: Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. CCS '12 (2012)