S³ Coursework Report: Automated ROP Exploit

https://github.com/jackbondpreston/security-cw

Jack Bond-Preston
jb17662@bristol.ac.uk

Liam Dalgarno
ld17285@bristol.ac.uk

Chris Gora
kg17815@bristol.ac.uk

1 Introduction

In this assignment we extended Return Oriented Programming (ROP) exploitation techniques introduced in Lab 4. This lab introduced a python tool called ROPGadget[1]. We modified ROPGadget and built a wrapper around it. This allowed us to make the ROP attack far more automated. We implemented the following:

- Automated offset detection.
- Arbitrary arguments to the `execve` syscall.
- Detecting and mitigating the presence of null bytes in addresses in the data segment.

2 Background

Return Oriented Programming [1] is at its core a stack buffer overflow vulnerability. This happens when an attacker writes more bytes to memory than can fit into the size of the buffer, hence placing arbitrary data in the stack frame (often referred to as "smashing the stack") [2]. If the overflow is large enough, it will extend to the location of the saved return address of the currently executing function. As a result, a function can be forced to return to any arbitrary place in memory, rather than to its caller. The stack still remains non-executable in this scenario. The attack has to be formed using existing instructions which are followed by `ret` – these are known as gadgets [1]. The `ret` instruction will pop the next instruction address off of the stack and into the Extended Instruction Pointer (EIP), allowing for chaining of gadgets. A series of gadgets that is used to execute code is referred to as a ROP chain. ROP gadgets can be found and searched automatically [3], even in some cases "blindly" (remotely and without access to the executable's binary content) [4].

This is extremely powerful - with a sufficiently large program, and thus sufficiently large pool of available gadgets, ROP chains are Turing complete [1, 5]. Thus, it follows that any shellcode can be executed through ROP [6, 7].

The method of ROP exploitation used in the lab was highly manual. It was the user's responsibility to find the correct padding length to overwrite the EIP. ROPGadget is also limited to only spawning a shell using an `execve("/bin/sh")` syscall. It therefore hard-coded which gadgets it looks for. It was not possible to automatically execute any other type of shellcode or syscall. This means that this tool can only produce very limited ROP chains.

2.1 Existing Tools

On the other hand, there is a range of other tools which provide more advanced capability. A small selection is discussed below:

pwntools[2] - This is a very extensive tool aimed towards Capture-The-Flag (CTF) competitions. It can analyse a binaries for various architectures and then automatically generate a ROP chain.

Ropper[3] - This is a tool similar to ROPgadget, but with a few more features. It has the ability to generate a ROP chain for arbitrary `execve` arguments (along with a few other predefined shellcodes). It is also effectively a search engine for various ROP gadgets. However, rather than searching for raw assembly, its unique capability is semantic search. It has a basic understanding of the actual semantics of gadgets, and their side effects. The user can specify a constraint such as `eax == 1 !ebx`. This will find all gadgets that can assign number 1 to `eax` while making sure that `ebx` isn't clobbered. It will also show a list of clobbered registers for each result of a general search, again utilising the semantic analysis of the gadgets.

ROPC[4] - A proof-of-concept compiler which creates ROP chains from an arbitrary shellcode written in a custom high-level language called ROPL.

[1] https://github.com/JonathanSalwan/ROPgadget

[2] https://github.com/Gallopsled/pwntools

[3] https://github.com/sashs/Ropper

[4] https://github.com/pakt/ropc

```
aaaabaaacaaadaaaeaaafaaagaaahaaa
```

Figure 1: Output of `cyclic(32)`.

3 Design & Implementation

Throughout the steps of this coursework we only target ELF x86 (32-bit) binaries. These were compiled on a Vagrant-backed Ubuntu `18.04` VM (Kernel `4.15.0-124`) with GCC `7.5.0` – statically linked (`-static`) and with the stack protector disabled (`-fno-stack-protector`).

3.1 Step 0

Originally, ROPGadget generated a Python file which had to be modified by the user (fixing the syntax, setting the padding, etc.) This created an unnecessary extra step for our task of creating an automatic tool. Due to this, we opted to modify ROPGadget to directly save the ROP string to a file.

3.2 Step 1

One of the main shortcomings of ROPGadget is that it requires a user to manually add padding of the correct length, so that the saved return address can be overwritten. We automated this process.

In our main utility Python file (`autoRop.py`), we generated a special input string using `pwnlib.util.cyclic`, initially of length 32 – an example of which is demonstrated in Figure 1. Each set of 4 bytes in this string is unique. We then ran the executable we were analysing using this string as input. We repeated the process, each time doubling the length of the input string, until the program crashed or we hit a configurable limit (at which we assumed the buffer overflow to not be viable). A crash implied that the buffer overflow reached and changed the `EIP`. We then read the last value of `EIP` from the core dump, using `pwnlib.elf.corefile`. Due to the uniqueness property of our input string, we used this value to determine the exact offset required for the padding.

A problem with this method is that it relies on core dumps, which may not be generated for a variety of reasons. Most notably, the kernel parameter `fs.suid_dumpable` [8] must be set to 1 or 2 to generate core dumps for setuid programs. This is insecure and therefore unlikely to be set (and is not by default). Without support for setuid programs, privilege escalation attacks through ROP would not be possible. Therefore, as an alternative, we simply brute force the offset. This is achieved by preparing a ROP chain which executes '`/bin/echo "[ ROP Successful! ]"`', and then incrementing the padding length until the ROP chain is executed. This method does not rely on reading the `EIP` from the core dump, therefore it is more versatile. We subsequently modified the ROPGadget code to accept the generated padding length, with the command-line flag `--paddingLen`.

```
["/bin/netcat", "-lnp", "5678", "-tte",
 "/bin/sh"]
```

Figure 2: An example `rop_exec.json` file.

Another thing we considered is that programs generally read input in the following ways: command-line arguments, reading from a file, and reading from `stdin`. We supports all three of these methods to pass the payload, by changing the command line argument `--input_method`. When passing the payload as a program argument, other arguments and the placement of the payload at a custom argument position are supported. It reads a file containing the argument layout, in this case it is simply:

```
[ "$PAYLOAD$" ]
```

where `$PAYLOAD$` is programatically replaced with the generated ROP chain.

This now gives us an automated workflow for generating and executing the ROP attack. The `autoRop.py` script does three things:

1. Find the padding length necessary to overwrite `EIP`.
2. Execute the customised ROPGadget script with this padding length to generate the ROP chain for the target executable.
3. Execute the target vulnerable binary with the generated ROP chain as its input. This binary can optionally be executed in an interactive mode.

3.3 Step 2

Originally, ROPGadget generated a ROP chain for a predefined `execve("/bin/sh", ["/bin/sh"], NULL)` shellcode. `execve` takes three arguments: `const char *pathname`, `char *const argv[]`, and `char *const envp[]`. In this step we added the ability to modify both the `pathname` and `argv` arguments. `execve` is a syscall, which means that it is invoked with `int 0x80`. On Linux x86, arguments are stored in registers in the following order: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`, where `eax` stores the unique syscall identifier. Therefore, we store `pathname` in `ebx`, and `argv` in `ecx`.

We opted to pass the custom `pathname` and `argv` into ROPGadget using a JSON file. This is because parsing arguments manually creates issues, for example with quotation marks or spaces. The JSON file contains an array of strings consisting of the executable to run and arguments to pass to it (with implicit `argv[0] = pathname`). An example of this is shown in Figure 2.

We modified the ROPGadget code to allow the writing of an arbitrary `pathname` string. ROPGadget uses the `.data` segment as a 'pseudo-stack', which it can write to using `mov [<reg>] <reg>` gadgets as demonstrated in Figure 3. Generally, the source data is popped from the stack into the register `src`, then the target address is popped from the stack into `dst`.

```
pop eax; ret;
pop edx; ret;
mov [edx], eax; ret;
```

Figure 3: Example gadgets which can be used to write to an arbitrary address in memory.

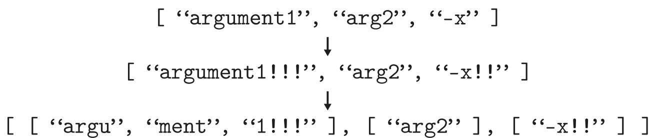

Figure 4: An example of argument chunking.

Finally, the data is written to the target address using `mov [dst], src`. Since the data being written is stored in 32-bit registers, it must be written in blocks of 4 bytes. In order to achieve this, we pad the string so that its length is a multiple of four. We use forward slashes (`/`) at the front of the string as the padding. For example, the executable name `"/bin/sh"` is written as `["//bi", "n/sh"]`. At the end, we use gadgets to append four null bytes to terminate the string.

We then allowed the execution of `execve` with an arbitrary `argv` argument. As opposed to the `pathname` padding, there are no "safe" characters that can be appended or prepended to make the arguments have a length divisible by 4 – for example, appending or prepending spaces does actually change the operation of many target programs. To give us strings for which all lengths are divisible by four, we append exclamation marks (`!`) to the end of each argument as placeholder characters. We then chunk each argument into strings of length 4. An example of this process is illustrated in Figure 4.

Each argument is written chunk by chunk. We then write 4 null bytes at the location of the first placeholder `!`. This is to overwrite the placeholder characters and terminate the string at the correct location. Now that the argument strings are in memory, we construct the `argv` array. We first write the address of `pathname`, followed by the addresses of each individual argument string. 4 null bytes are then appended, as the `argv` array must be null terminated. An example of this data in memory is shown in Figure 5. The address of this array in memory can then be passed in `ecx` to `execve`.

The user may execute any program that they like, and some programs may be interactive, such as a shell. For this step, we added a command line argument `--interactive` which uses `pwnlib.tubes.interactive` to attach to the program and emulate an interactive terminal. Without this argument, our tool simply dumps the output of the program to `stdout`.

3.4 Step 3

The original ROPgadget tool could fail when the data segment addresses it writes to contained null bytes. This was because when the ROP chain input is being treated as a string, any null bytes would be seen as the null terminator of said string. We fixed this by checking if the start of the data segment is even. If it is, we add 1 to make it odd. This ensures that the data addresses we write to will not end with a null byte. This is the most likely place for a null byte to appear, because the least significant bits change as we write our data but the most significant are not likely to unless the input is very large.

We also check if the data segment address contains a null byte at any location other than the end. If it does we replace the `0x00` with a `0x01`, such that we change the address by the minimum possible amount while still removing the null byte.

We ensured this solution was working correctly by compiling a test program using the linker option `-Tdata <address>` to set the data segment start address to various addresses containing null bytes. The tool proved effective against these data addresses.

3.5 Step 4

We did not implement step 4. We did extensive research on potential solutions to it, however, we were not able to come up with a solution reasonable to implement in the given time frame. Below we discuss existing successful solutions as well as our own ideas and the issues surrounding them.

Shell-storm[5] contains a repository of known working shellcode for a variety of architectures. We first analysed these to get an idea of the general structure of shellcode. A key assumption of shellcode is that it is placed in an executable segment of memory and thus free to use *any* instruction. Therefore, shellcode can modify registers freely, while ROP is limited by gadget register assignment and clobbering. This severely limits how ROP can use registers because register usage in programs is quite limited: registers tend to be used for specific purposes, such as `eax` for arithmetic operations and `ebx` for pointing to memory locations. Converting a shellcode's free use of registers to gadgets requires analysis of which registers are currently available and which registers a gadget clobbers.

Furthermore, in ROP, we cannot freely push to and pop from the stack. Our ROP relies on the stack containing very specific data, to ensure `ret`s will return us to the correct location. To be able to execute arbitrary shellcode, it is necessary to emulate the stack in the `.data` segment. This can be done by converting all `push` instructions to a write to memory, and `pop` instructions to a read from memory. A virtual stack pointer can be kept track of by the compiler to know where to write to/read from, and offset accordingly after each stack operation. One problem this causes is that it consumes too many registers. A normal `push`/`pop` only involves one general purpose register. This memory stack approach requires at least two of our usable registers – one to hold the destination memory address, and one to provide/receive the value. This

[5] http://shell-storm.org/shellcode/

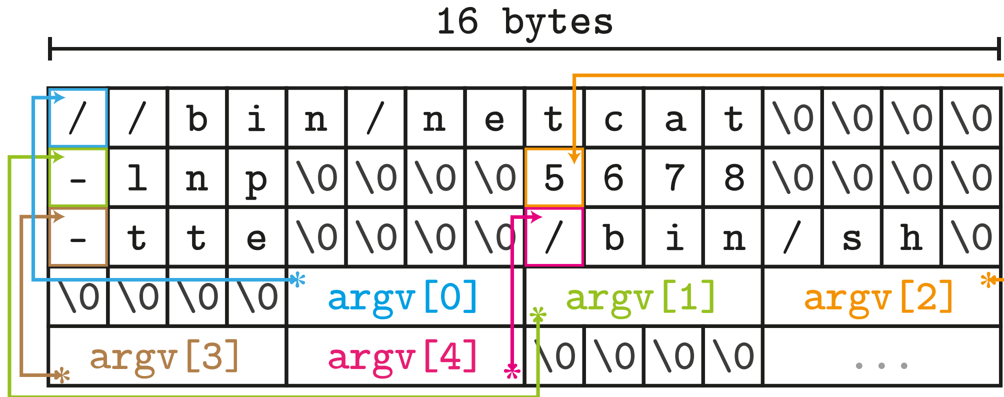

Figure 5: An example `.data` layout for the `netcat` remote shell ROP chain. Empty cells indicate unknown and unneeded contents (never manually overwritten).

is also the reason that banking registers to aid in register management actually ends up being detrimental, as banking one register involves overwriting another with a memory address.

One existing solution, fuck-riscv-rop[6], implements a Brainfuck to ROP compiler [7], allowing for Turing-complete ROP computation. A buffer in memory is used to implement Brainfuck's tape. However, this relies on a "self-modifying ROP" technique to deal with register clobbering issues by saving/restoring registers around these gadgets, which is specific to the RISC-V architecture. It also only targets `glibc-2.30.9000-29.fc32.riscv64` specifically.

3.5.1 Q

Schwartz et al. show an implementation of compiling arbitrary shellcode into a ROP chain [9]. It uses a custom high-level language called QooL to define the shellcode. Q claims to be able to effectively generate ROP chains for 80% of Linux binaries greater than 20KB in size.

Q works by analysing the semantics of gadgets. Instead of searching for predefined gadgets (e.g. `mov eax, ebx`), it searches for groups of instructions for which the semantics are useful (e.g. `imul 1, eax, ebx` is a move gadget). This allows it to find a much larger selection of gadgets, which gives a lot more flexibility in target binaries. Each gadget is identified with a specific type (e.g. certain types of moves, or memory stores). This is defined by a postcondition which must be true any time the gadget is executed.

Once the gadgets in the target binary have been identified, a complex compiler is used to assign and arrange them into a working implementation of the QooL program. This process is very involved due to all the interdependencies of different gadgets, and the limited amount of register space (and inability/ineffectiveness of banking them).

The Q source code is not available, however the paper claims that the program is 4585 lines of OCaml code. ROPC (as mentioned in Subsection 2.1) is an implementation based on Q, which comes in at 4675 lines of OCaml code. Both of these were written by highly accomplished experts. Thus, we did not consider it possible to complete an implementation of Q for this coursework.

[6] https://github.com/garrettgu10/fuck-riscv-rop

4 Evaluation

4.1 Automation

Our aim of automating the ROP attack has been successful. Compared to ROPGadget, our tool is able to generate complete ROP chains for any arbitrary `execve` syscall with any `argv` argument. Unlike ROPGadget and Ropper, we can automatically calculate the correct length of the padding so that the EIP is overwritten. Executing a ROP exploit with our tool is a one-step process: the user simply has to provide a binary and we will automatically prepare and execute the attack. The aforementioned tools require the attacker to perform this step manually. Our range of automated ROP features and automation is comparable to the offering of pwntools (discussed in Subsection 2.1).

4.2 Arbitrary Execution

The ability to execute an arbitrary `execve` syscall is the most important improvement on ROPGadget. It means that the range of attacks is much greater and it offers many more opportunities to the attacker. An example of this would be where the target program is not accessible via. an interactive shell. Here executing `/bin/sh` would not be a useful attack, as the attacker would have no access to the executed shell. With our solution, a remote shell can be launched by choosing to launch `netcat` (or similar) with `execve`.

4.3 Non-Executable Stack

Modern systems such as Linux use a strategy often referred to as W⊕X (writable XOR executable) and mark each page of memory either as executable (and thus read-only) or as non-executable (and potentially writable) [1]. The stack is always writable, thus cannot be executable. By definition, a ROP attack does not require an executable stack (or any writable executable memory), as it re-uses the existing program code in the (executable) text segment without needing to modify it and can therefore bypass any hardware non-executable stack protections (such as the Intel XD bit and AMD NX bit) [10, 11, 1].

4.4 Target Support

There are certain flaws with our project when compared to the open-source alternatives. We currently limited the scope of our ROP chain generation to Intel x86 ELF binaries. This acts as a proof of concept which could then in future be translated to other executable types and architectures. ROP attacks have been successfully demonstrated on other architectures such as Arm [12], RISC-V [7] and SPARC [13], as well as other binary formats such as Windows PE binaries [6].

We tested our program on the following selection of target binaries:

1. `null-data-addr`: A copy of the vulnerable program from the lab which is compiled with `-Tdata 0x080f0000`. This program takes a file name as its only argument, and the buffer overflow exploit occurs upon reading this file (which we place the ROP chain into). Our program successfully mitigates the null bytes in the data address and the generated ROP chain is successful.

2. `elf-Linux-x86` / `elf-Linux-x86-NDH-chall`: These are 32-bit Linux executables from the `test-suite-binaries` subdirectory of ROPGadget. They take a string argument, and this has a buffer overflow vulnerability. Our program is able to successfully find the offset and execute a ROP chain.

3. `crashmail`: As a real world example, we took a vulnerable version (1.6) of Crashmail II[7]. This has a buffer overflow vulnerability when the `SETTINGS` option is used. We set up our `exec_args.json` file as follows:

   ```
   [ "SETTINGS", "$PAYLOAD$" ]
   ```

   and the exploit is successful!

Our testing shows that our program is able to find the offset and generate an appropriate ROP chain for an array of vulnerable programs, which take the exploit input in different ways (argument string, payload file, stdin).

4.5 Stack Protector

Our solution is not effective if Stack Smashing Protection (SSP) was enabled when the target binary was compiled. For all our test binaries we compiled with the `-fno-stack-protector` flag to disable this.

The stack protector prevents our ROP exploit via the following mechanism [2, 14]:

1. A canary value is added at the end of the local variables in the stack (before the EIP). This value is unknown to the attacker.
2. At the end of the function call, before returning to the saved EIP, the canary value's integrity is checked. If it has been modified, execution will halt and the attack is unsuccessful.

In the case of our ROP attack, we cannot overwrite the EIP without modifying this canary value, thus the attack will not work.

4.6 Address Space Layout Randomisation

Our attack is ineffective with ASLR enabled for the target binary in Linux. The test binaries used were all compiled with `-static` which disables the generation of a Position Independent Executable (PIE) and thus relocation of the `.text` segment for this program (some things, such as the library code, are always randomly relocated [15]).

ASLR prevents the success of a ROP attack by making the memory locations of the ROP gadgets differ run to run [11, 10]. This means we cannot hard-code these locations into our ROP chain. ASLR must be enabled for every module used in the target, lest a ROP exploit be formed around those for which it was not [16].

Even ASLR is not infallible – a memory disclosure exploit can be used to map the memory locations of gadgets even with ASLR enabled and generate a ROP chain on the fly [17]. Targets will not necessarily always be PIEs due to the performance cost [18], or lack of implementation.

[7] https://www.exploit-db.com/exploits/44331

5 Appendix

5.1 Objectives of the Proposal

We successfully completed objectives 1-3 identified in Subsection 3.2-Subsection 3.4.

We were not able to complete Step 4 of our original proposal (as detailed in Subsection 3.5), however we did research the area extensively and learnt a lot. We made attempts to start part 4, mostly consisting of reasoning about and planning a solution. We found that it was just too complex for us to be able to complete it in the given timeframe, and ended up mostly focussing on researching the issue and understanding how it could be implemented with more time. We tried to think of a limited solution to this task, but we could not think of a restricted feature set that would work properly – the task felt all or nothing if it was to work on multiple binaries and shellcodes.

5.2 Individual Contributions

We had an issue in the first week with illness - two of the group members (who live together) fell ill with mild flu-like symptoms. Whilst we were still able to work, it was at a slower pace and for fewer hours each day. After the first week, everyone was in good health again.

We did not find the programming task to be easily parallelisable. We worked together using VS Code Live Share on the programming. This ensured we all had a complete understanding of the solution, and enabled us to combine our differing knowledge areas. Research and report writing was conducted more in parallel, but each member again made an equal contribution.

Thus, all members made an equal total contribution to the project:

- Jack Bond-Preston: 33%
- Liam Dalgarno: 33%
- Chris Gora: 33%

6 Glossary

AMD NX AMD's No-eXecute bit technology preventing memory from being both writeable and executable. Also known as Enhanced Virus Protection (EVP).

clobber To modify a register that you did not specifically want to modify. Often caused by a gadget having extraneous instructions to the instruction(s) you are utilising the gadget to execute.

gadget A select sequence of instructions followed by a return, from the target program. These are chained together to execute useful code.

Intel XD Intel Execute Disable. Intel's No-eXecute bit technology preventing memory from being both writeable and executable.

setuid A Unix access flag which can be set on files, directories, and executables. If set on an executable, users will execute it with the privileges of the owner.

shellcode A malicious code payload, often with the aim of spawning a shell for the attacker to abuse.

7 Acronyms

ASLR Address Space Layout Randomisation.

EIP Extended Instruction Pointer.

ELF Executable and Linkable Format.

PE Portable Executable.

PIE Position Independent Executable.

ROP Return Oriented Programming.

SSP Stack Smashing Protection.

8 References

[1] Hovav 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*. CCS '07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 552–561. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313.

[2] Aleph One. "Smashing the Stack for Fun and Profit". In: *Phrack* 7.49 (Nov. 1996). URL: http://www.phrack.com/issues.html?issue=49&id=14.

[3] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. "A Framework for Automated Architecture-Independent Gadget Search". In: WOOT'10. Washington, DC: USENIX Association, 2010, p. 1.

[4] A. Bittau et al. "Hacking Blind". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 227–242. DOI: 10.1109/SP.2014.22.

[5] "Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming". In: *6th USENIX Workshop on Offensive Technologies (WOOT 12)*. Bellevue, WA: USENIX Association, Aug. 2012. URL: https://www.usenix.org/conference/woot12/workshop-program/presentation/Homescu.

[6] C. Ntantogian et al. “Transforming malicious code to ROP gadgets for antivirus evasion”. In: *IET Information Security* 13.6 (2019), pp. 570–578. DOI: 10.1049/iet-ifs.2018.5386.

[7] Garrett Gu and Hovav Shacham. *Return-Oriented Programming in RISC-V*. 2020. arXiv: 2007.14995 [cs.CR].

[8] Rik van Riel and Shen Feng. *Documentation for /proc/sys/fs/**. 2009. URL: http://web.archive.org/web/20200708133511/http://www.kernel.org/doc/Documentation/sysctl/fs.txt (visited on 12/05/2020).

[9] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “Q: Exploit Hardening Made Easy”. In: *Proceedings of the 20th USENIX Conference on Security*. SEC’11. San Francisco, CA: USENIX Association, 2011, p. 25.

[10] Sebastian Krahmer. “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique”. In: (Oct. 2005). URL: https://users.suse.com/~krahmer/no-nx.pdf.

[11] H. M. Gisbert and I. Ripoll. “On the Effectiveness of NX, SSP, RenewSSP, and ASLR against Stack Buffer Overflows”. In: *2014 IEEE 13th International Symposium on Network Computing and Applications*. 2014, pp. 145–152. DOI: 10.1109/NCA.2014.28.

[12] Tim Kornau et al. “Return oriented programming for the ARM architecture”. PhD thesis. Master’s thesis, Ruhr-Universität Bochum, 2010.

[13] Erik Buchanan et al. “When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC”. In: CCS ’08. Alexandria, Virginia, USA: Association for Computing Machinery, 2008, pp. 27–38. ISBN: 9781595938107. DOI: 10.1145/1455770.1455776.

[14] Bruno Bierbaumer et al. “Smashing the Stack Protector for Fun and Profit”. In: *ICT Systems Security and Privacy Protection*. Ed. by Lech Jan Janczewski and Mirosław Kutyłowski. Cham: Springer International Publishing, 2018, pp. 293–306. ISBN: 978-3-319-99828-2. DOI: 10.1007/978-3-319-99828-2_21.

[15] Hector Marco Gisbert and Ismael Ripoli. “On the Effectiveness of Full-ASLR on 64-bit Linux”. English. In: In-depth Security Conference 2014 (DeepSec) ; Conference date: 18-11-2014 Through 21-11-2014. Nov. 2014. URL: https://deepsec.net/archive/2014.deepsec.net/index.html.

[16] Nicholas Carlini and David Wagner. “ROP is Still Dangerous: Breaking Modern Defenses”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 385–399. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[17] K. Z. Snow et al. “Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization”. In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 574–588. DOI: 10.1109/SP.2013.45.

[18] Mathias Payer. *Too much PIE is bad for performance*. en. Tech. rep. Technical Reports D-INFK. Zürich, 2012. DOI: 10.3929/ethz-a-007316742.