COMSM0102 Coursework: Automating Return Oriented Programming Exploits

Karl Sachs
eh19257@bristol.ac.uk
University of Bristol

Joseph Eastoe
ur19476@bristol.ac.uk
University of Bristol

December 15, 2022

1 Introduction

In this report we automate the Return Oriented Programming techniques that were first discovered in the 1990s and investigate the power of them and as well as their shortcomings. All 4 tasks at hand were reached with high standards for 32-bit binaries on a x86 architecture. `ROPgadget.py` [Sal15] was used as a backbone to find gadgets in the vulnerable programs that are located in the code base, it was then modified so that it can produce ROP chains that execute arbitrary `execve` syscalls or arbitrary shellcode. Our repo can be found here [SE22].

2 Background

Return Oriented Programming (ROP), is a type of exploit that allows an attacker to execute arbitrary code on the target system by reusing small snippets of code called "gadgets". These are already found in the system's memory and chaining them together to achieve the some chosen execution [Sha07]. Each gadget has a desired operation (e.g `pop ecx`) that ends with a return instruction (or any other control flow instruction). A gadget ending in `ret` will causes the program to `pop` an address off the stack and into the EIP, running it. Populating the stack with addresses of gadgets will lead to multiple gadgets being called and executed sequentially. Using only a small variation of gadgets in combination with each other, one can create a Turing complete machine [HSL+12].

Initiating the execution of a ROP chain can be done by overwriting the return address of a function with the address of the first gadget in the chain - this can be done using a buffer overflow exploit [One96]. A buffer overflow occurs where a function that has no bounding check is used (normally takes in user input) and writes more data to a memory buffer than it is allocated to store. This causes an overflow of the buffer overwriting important values stored on the stack etc. the return address used to return to the previous frame [One96]. Thus one can use a buffer overflow to store the ROP chain onto the stack and cause the program to return to first gadget of the ROP chain initiating the chain.

The advantage of using ROP over a classical buffer overflow is that it can bypass "write-xor-execute" - a security property that describes memory pages that are either writable or executable, but not both. The stack has to be writable thus it must not be executable using this rule. This protection prevents buffer overflow and heap overflow attacks from gaining easy code execution by writing shellcode to the stack or heap and overwriting the return address to point to the shellcode, thus gaining code execution. However, ROP as a technique, does not require pages to have both write and execute access protection and so "write-xor-execute" does not mitigate ROP attacks.

A common prevention method to stop ROP based-attacks is to use Address Space Layout Randomization (ASLR). ASLR randomises the memory addresses used by a program thus an attacker does not know where the gadgets lie as they change address each time the program is run.

2.1 Existing Tools

Many tools exist for finding gadget and generating ROP chains with ropper [Sas14], ROPgadget [Sal15] and ropshell.com [rop14] being some notable ones. However, sometimes they fail to find the desired gadgets for example the gadgets in __libc_csu_init() for dynamic link binaries. Ret2csu uses the gadgets in __libc_csu_init(), that can even be found in dynamically linked executables with its main function just being a return, to form what they call a universal µROP chain. Using these gadgets an attacker can leak arbitrary addresses (for

example libc), then generate a ROP chain now using the whole of libc - this has enough gadgets to be Turing complete [MGR18]. One thing to note is that this would require the program to take in multiple sets of user input using the same random addresses. Numerous program runs with consistent random addresses can leak information about the gadgets in libc, this information can then be used to create ROP chains. In order to keep consistent, random addresses an attacker would need to make sure they don't crash the program. This method was successful in creating a ROP chain on a fully-protected, 64-bit executable, bypassing such protections like ASLR, W $\bigotimes$ X etc.

Out of the tools listed, we used ROPgadget. It works by using Capstone [Cap20] to disassemble the binary, searches through this finding operations that end in a control flow instruction and forms a list of gadgets. If the flag `--ropchain` is set then depending on the architecture, the file ropmakerx86.py or ropmakerx64.py are run. These look for gadgets listed here: `(xor eax, eax)`, `(inc eax)` , `(pop ebx)` , `(pop ecx)` , `(pop edx)` , `(int 0x80)` , `(pop SRC)` , `(pop DST)` , `(xor SCR)` and `(mov dword ptr [SRC], DST)` where SRC and DST are general registers (SRC and DST can't equal the same register). `ROPgadget`'s `mov`, `(pop SRC)` and `(pop DST)` gadgets can be thought of as semantic gadgets, this is because they make use of arbitrary registers and not specific general registers. As a whole, these gadgets are all that's required to implement a ROP chain that performs the system call `execve("/bin/sh", NULL, NULL)`.

3 Task 1 - Auto-Padding

With any buffer overflow exploit, one first needs to find the amount of padding that is needed such that the return address of the current function can be overwritten; `auto-padder.py` can automatically calculates the padding length needed. It takes a program - assumed to be vulnerable to a buffer overflow - and finds it by increasing the length of a specially crafted string which becomes the input of the vulnerable program. A program can take input in three different ways: through a file, as an argument or via the input stream `STDIN`. `auto-padder.py` can find the amount of padding for all 3 types of input using the flags `--file`, `--arg` or `--stdin` respectively. This aids in running an ROP exploit on a wider range of executables.

Under the hood, `auto-padder.py` makes use of `strace`[Kra20]- a tool that is used to "trace syscalls and signals"[Kra20] - to run the vulnerable program. On output, `strace` will tell us when a program exits and why it exits. A buffer overflow will cause the program to crash due to a segmentation fault, this is where a program attempts to access undefined memory. `auto-padder.py` can detect this, look at the flagged address and see if it matches the end of the specially crafted string that was used as input.

This string that `auto-padder.py` generates importantly ends in `!!!!` - this is then used as the input for our the vulnerable program. Until a segfault occurs in the program, the string is pre-appended with `0x07 0x07 0x07 0x07` in order to increase its length. On a segmentation fault, `strace` will provide the address that caused the fault. We continue prefixing the string with `0x07 0x07 0x07 0x07` until `strace` flags a segmentation fault where the address that caused it is equal to the head of our string `0x21212121` (equal to !!!! in hex). The amount of padding that we need to exploit this buffer overflow is equal to the number of `0x07` bytes. We pre-append 4 bytes to our string per iteration instead of 1 as the buffer will always use an amount of memory that is a multiple of the word-size, in this case we have a machine with 32-bit words. This slightly improves the performance of `auto-padder.py`.

A different method could be to create a string that has a different character every 4 bytes, i.e. `01 01 01 01 02 02 ....` We could then double the length of the string ensuring there are unique characters in chunks of 4 bytes - we opted not to use this method as although it is faster to run, the size of the string is limited to to the number of different bytes available with exception of `NULL` bytes (`0x00`) and line feed (`0x0A`) where the input type is STDIN. Generally, finding all possible buffer overflow vulnerabilities in a program was considered more important that computational efficiency of our this particular program.

4 Task 2 - Arbitrary Command Execution

`ROPgadget.py` [Sal15] itself is able to create a ROP chain but only for `execve("/bin//sh", NULL, NULL)`. In order to run an arbitrary `execve` we need to be able to control the syscall's parameters: `*filename` and `*argv`. In order to have more control we need to be able to control`*envp`. In linux, syscalls have their own calling convention which is different from the C calling convention, the requirements for each linux syscall can be found here: [Chr22].

Each element of *argv needs to point to a

string that is terminated by a NULL byte (0x00) otherwise the syscall will not know where the argument ends, we cannot put a NULL character directly into the ROP chain as the function strcpy(), that copies our ROP chain into memory, will terminated on receiving a NULL byte; we therefore need to generate NULLs at runtime. NULLs need to be placed at the end of each string to terminate it as well as at the end of *argv as the array of pointers is also required to be NULL terminated. NULL bytes can be generated in a manor of ways but we opted to do it by XOR'ing the SRC register with itself - this will result in a register full of 0s. Using ROPgadget's generic write4where gadget, we can load the address of the end of the string into DST, write NULL bytes there and terminate the string. This method of NULL-terminating strings requires a small amount of gadgets and so will not increase the length of the ROP chain significantly; it also is computationally faster than other methods of loading in a NULL bytes as an XOR is a fast, hardware-accelerated, bitwise operation.

Using arguments with arbitrary lengths means that when they are written into memory, arguments that follow might not be packed into memory correctly, causing the front of them to start misaligned. Depending on the accessing granularity of the system, one may not be able to point to the start of the string. We can simple overcome this issue by padding strings such that they are multiples of word size and therefore start at the begin of an address word. If an argument contains a path, we can simple replace a '/' with '//' - this has the same semantic meaning for a file path. For arguments that are not file paths, we can pad the end of string with 0x07 bytes until the size of the string is a multiple of the word size - this will make the next string that's loaded into the .data section of memory aligned. During runtime, the ROP chain can be encoded to write a NULL character to the first 0x07 byte that is found in the string, NULL terminating it at the correct place. This retains the alignment of the argument strings in the .data section of memory. 0x07 is the ASCII escape character for a bell, we opted to use this to pack the strings and in section 3 as (i) it cannot be typed on a standard "QWERTY" keyboard and (ii) it is a very rare character to come across as it is not used anymore; both these reasons mean that it is very unlikely that this character would be used in a execve - if it were used then it could cause the NULL character to be written to the incorrect spot, changing the arguments of the syscall. We therefore want an unlikely character to be used for this job.

Our modified version of ROPGadget.py takes in a user-defined execve from the environment variable ROPCMD.

5 Task 3

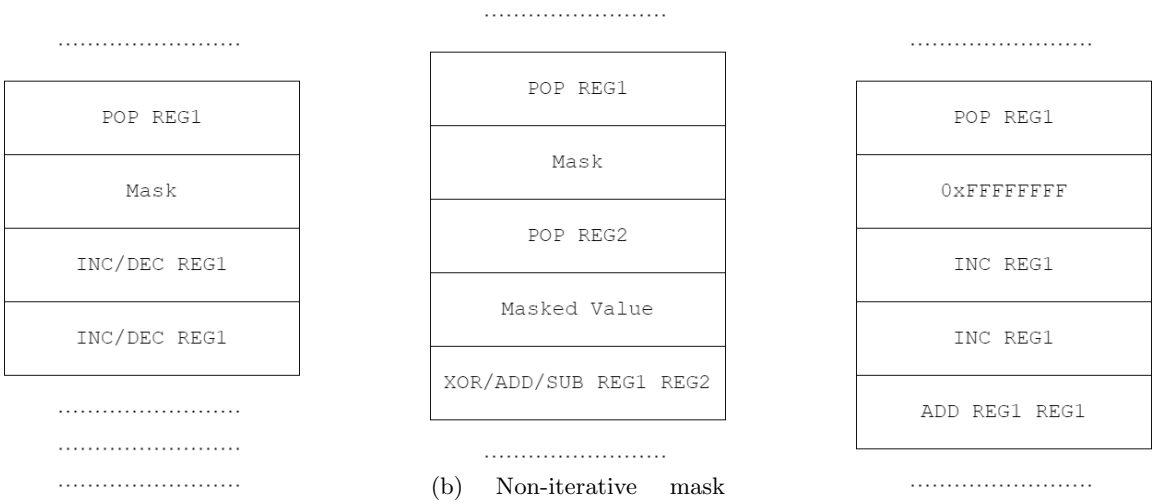
As previously mentioned, NULL bytes are the bane of any ROP-based attack as it means the full chain cannot be copied into the program's memory. Handling .data addresses that contain NULL bytes in the ROP chain is done by first encoding the values during the ROP generation and then decoding them during the runtime of the exploit - this can be implemented in multiple ways. To encode a NULL containing .data address, we created a mask and a masked address from it - we can then combine these with a masking gadget such that the value is decoded when it is ran. A major hurdle in the project was the lack of available masking gadgets in our small test programs; to overcome this we made use of multiple masking gadgets which can be divided into two groups: iterative masks and non-iterative masks.

Iterative masks consisted of inc and dec gadgets - during runtime, a masked address is popped into a register, this was then followed by n iterations of inc or dec. The mask chain for this is visualised in figure 1a.

Non-iterative masks include xor, add and sub gadgets. A mask and a corresponding masked address are combined together using one of the aforementioned gadgets to produce the unmasked value in some output register - the mask chain for this can be visualised in figure 1b. In most cases, non-iterative masks are more spatially efficient and were therefore more favoured by our modified version of ROPgadget.py than an iterative mask. Having a large set of possible masks at disposal means that even very small programs that do not have a large variation of gadgets are still vulnerable to this ROP-based attack.

This masking technique also allows for gadgets that contain NULL bytes to be run. Although quite rare in a 32 bit architecture, NULL bytes can be found in gadget addresses and can be dealt with using the above masking and unmasking techniques with the only addition being the value is push to the stack after - this is supported in our modified ROPgadget.py. This would be very useful when creating ROP chains for a 64-bit architecture as most addresses are prefixed with NULL bytes.

In some cases of the iterative masks, the collection of masks that we had would generate ROP chains that were over 17,000 bytes long which is not practical. In theory, if we only have an



(a) Iterative mask chain.

(b) Non-iterative mask chain.

(c) Double then increment.

Figure 1: The 3 different types of mask chains.

`dec` gadget and we want to encode the number `0x00000000` then we would need to start at `0x01010101` and decrement from there - this would take a mask chain 16843009 bytes long excluding padding - this is definitely not practical.

This was overcome by developing another masking technique, here we combine a gadget that sums a register with itself (i.e. doubles it) along side an `inc` gadget. We can use the `inc` gadget to set the least significant bit of 0 to a one and then use the double gadget to left shift this bit. By having a chain of doubles and increments we can build a number up from `0x0` - this is visualised in figure 1c. We can load `0x0` into a register in many ways, for example by XOR'ing a register with itself or by loading in `0xFFFFFFFF` and incrementing.

Using only the `inc` gadgte to put the number `0x00001000` (4096_{10})into the ECX required a ROP chain of 16400 Bytes, in comparison using a combination of doubling the ECX and then incrementing it requires a ROP chain of 68 bytes.

6 Task 4

Originally, buffer overflow exploits were able to directly run shellcode - code that spawns a shell - located on the stack. To mitigate this, the "write-xor-execute" memory protection policy as discussed in section 2 was introduced. An adversary that is trying to run shellcode that they've inputted onto the stack won't be able to run it as the stack, itself, is non-executable. Our modified version of `ROPgadget.py` can overcome this mitigation by making using of the $mprotect(void* addr, size_t len, int prot)$ syscall which changes the memory protection rule to a combination of `READ`, `WRITE` and/or `EXECUTE` for the paramertised address. We first need to write our shellcode into the `.data` section of memory as done before in section 4. The address `addr` that is passed into `mprotect` must be aligned to the start of a page. The vagrant machine used has pages of size 4096 Bytes (`0x1000` Bytes) and so a page must start on an address that ends in `0xYYYYY000` (where Y are arbitrary hex values). Therefore we can find the start of the page that is before our shellcode by applying a bitwise AND operation with the start address of `.data` and `0xFFFFF000`. The memory was given read, write and execute access rights to show that it can bypass the "write-xor-execute" memory policy but in reality, it only needs the execute.

7 Evaluation

`auto-padder.py` can successfully find the amount of padding required to overwrite the return address of a function in a program that is assumed to have a buffer overflow vulnerability in. It can calculate the amount of padding required on programs that take input via a file, an argument or via STDIN, it cannot automatically detect which input method is vulnerable by itself - this needs to be selected by the user.

Arbitrary `execve`s, that are defined in the environment variable ROPCMD, can be successfully run as a result of the exploit, however it cannot generate a ROP chain for `execve` syscalls that use environment variables - the *envp always points to NULL. Further work could be done in order to implement this and provide the execution of more complex programs.

Our modified `ROPgadget.py` can successfully handle `NULL` bytes found in .data addresses as well as `NULL` bytes found in gadget addresses. The program makes use of 6 different types of masking techniques in order to find the spatially optimal mask chain for some `NULL`-containing value. It does, however, do this in detriment to the computational efficiency of the program. Making the ROP chain as small as possible is essential and the encoding of `NULL` bytes is one of the main reasons for large ROP chains. In small programs, where there is little variation in the number of gadgets, we can get very large mask chains when only iterative masking gadgets are available.

Finding the smallest mask chain requires the program to check the length of every possible one. Usually this is a fast process but in cases where their are a lot of chains and the chains themselves are long (i.e. iterative chains), this can take upwards of 50 seconds to complete (tested on a lab machine) when run using `vulnz/vuln4-32`. The program capped the number of `inc` or `dec` gadgets in an iterative mask chain to 500 in order to help with computational efficiency. This equates to a 2000 byte long mask chain excluding padding. Finding the smallest mask was key in making the ROP chain as small as possible as it means it is harder to detect and can be inputted into a larger pool of programs.

As a general note, both `ROPgadget.py` and `auto-padder.py` require little dependencies to run, this is useful as it means that they are lightweight and can be run on a greater number of platforms.

`ROPgadget.py` can generate a ROP chain that can successfully run arbitrary shellcode and is successful at by-passing the "write-xor-execute" security feature. It is to be noted that the shellcode does need to be `NULL` free however using the same technique as described in section 5, this could very easily be overcome - this was never implemented due to timing restraints.

7.1 Limitations

7.1.1 64-bit Executables

Our modified `ROPgadget.py` is only capable of generating successful ROP chains for a 32-bit x86 architecture and is unable to do this for 64-bit programs. Most 64-bit addresses start with `NULL` bytes, this means that we cannot create masking chains as the addresses of the masking gadgets contain `NULL` bytes themselves. An attempt to get around this was to use stack pivoting.

7.1.2 Stack Pivoting

Stack pivoting is a technique that changes where the stack pointer points to. This could be useful, for example, in conjunction with the fread function found in vuln3-64 (which is able to read in `NULL` bytes). fread could be used to place the payload onto the stack, then using the gadget at the end of padding, we could change the stack pointer, returning would now run the payload copied in by fread (which has the correct null bytes). We used ropper with the stack pivot flag set to find such gadgets. However we were unable to find such gadget, the most we could find is to return with an offset which does a normal `ret` and then changes the stack pointer. Code for this attempt can be found in `extensions.zip`.

The modified `ROPgadget.py` cannot generate ROP chains for programs that have ASLR turned-on. This is because `ROPgadget.py` needs to build an exploit and then run it in different instances of the vulnerable program. With ASLR turned on, a program randomises specific parts of its memory addresses - this means an attacker would not know where the addresses of the gadgets in the program are located.

8 Conclusion

This paper shows that `auto-padder.py` and the modified `ROPgadget.py` can, together, take a basic Return Oriented Programming technique that spawns a shell and can extend it such that for a given program vulnerable to a buffer overflow attack, they can automatically generate a string that can run an arbitrary `execve` or execute some arbitrary shellcode. It must be noted that this technique ceases to work when modern mitigation systems are used with the exception of the "write-xor-execute" data policy.

References

[Cap20] Capstone. The ultimate disassembly framework, May 2020.

[Chr22] ChromiumOS. Linux system call table, 2022. Last accessed December 11, 2022.

[HSL+12] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. *21st USENIX Security Symposium*, Aug 2012.

[Kra20] Paul Kranenburg. Strace(1), Nov 2020.

[MGR18] Héctor Marco-Gisbert and Ismael Ripoll. return-to-csu: a new method to bypass 64-bit linux aslr. 2018.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 11 1996.

[rop14] Free online rop gadgets search, 2014.

[Sal15] Jonathan Salwan. Jonathansalwan/ropgadget: This tool lets you search your gadgets on your binaries to facilitate your rop exploitation. ropgadget supports elf, pe and mach-o format on x86, x64, arm, arm64, powerpc, sparc and mips architectures., Apr 2015.

[Sas14] Sashs. Sashs/ropper: Display information about files in different file formats and find gadgets to build rop chains for different architectures (x86/x86 64, arm/arm64, mips, powerpc, sparc64). for disassembly ropper uses the awesome capstone framework., Aug 2014.

[SE22] Karl Sachs and Joseph Eastoe. Comsm0049: Repo for sss cw, Nov 2022.

[Sha07] 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, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery.

A Individual Contributions

A.1 Karl Sachs

Contribution weight - 55%

Over the coursework period we both managed a sizeable amount of work. From the assignment itself, I completed all of task 1 and produce the program `auto-padder.py` as a result of this. In task 2, I generate the code that `NULL` terminated strings in memory, as well as the code that padded the ends of arguments with enough bytes that they could be aligned in memory and (also `NULL` terminated). In task 3, I designed some early methods of unmasking values during runtime - these ended up being flawed. On top of this I iterated on the design of Joe's double with incrementing mask generator as well as creating the code which produced the masks for iterative and non-iterative masks. I implemented the majority of task 4.

In terms of the report, I fully wrote, proofread and finalised sections 1, 3, 4, 5, 6, 8 and the appendix B and C. I contributed to half of the content found in section 7.

I wrote the script for, recorded and edited the demonstration as well as wrote the README.md found with the code.

A.2 Joseph Eastoe

Contribution weight - 45%

Task 2 100%: - Discovering where to add code in ROPgadgets - Writing a execve with args using ROPgadgets gadgets

Task 3 75%: - Made the program find possible masks chains - Made the program use the mask chain which generates the shortest payload - tried to implemented Double with Increment however my method didn't work - bug fixes

Task 4 5%: - research about using mprotect - bug fixes

Research 100%: - research about ret2csu, reading the paper - research about stack pivoting trying use this for the exploit - research bypasses for x64

Report 25%: - section 2 - section 7.1.2

B Challenges

Overall, we achieved all 4 of the tasks successfully.

The main technical challenge that showed itself during this project was finding suitable gadgets during during task 3. This was difficult as we were testing our modified `ROPgadgetp.py` on small, vulnerable programs - this meant that the

gadget count, specifically for masking gadgets, was low. As discussed in 5, we overcame this issue by having a wide variety of masking gadgets at our disposal.

A lot of time went into researching the possibilities of generating ROP chains for a 64-bit architecture as well as expanding the program to be able to handle ASLR - this research was entirely carried out by Joseph. Both of these goals were not reached but stack pivoting, as a technique, was used in an attempt to break a 64-bit architecture - this was briefly discussed in section 7.1.2.

C Glossary

ROP chain - the chain of gadget addresses that is inputted into the vulnerable program. An adversary has control of this and can use it to run what they wish.

Mask Chain - a subsection of the main ROP chain, that is used to unmask a value during runtime of the exploit.

C.1 Masking Gadgets

Iterative - a masking gadget that uses either an `inc` or `dec` gadgets to decode some value.

Non-Iterative - a masking gadget that uses either an `xor`, `add` or `sub` gadgets to decode some value.

Double with Increment - a masking gadget that uses a combination of `inc` gadgets with a gadget that adds to itself in order to decode some value.