ROPOverflow: Buffer Overflow ROP-Based Exploit Generator Using Arbitrary Shellcode

Ruairi Fox
University of Bristol
rf17160@bristol.ac.uk

Bogdan Stelea
University of Bristol
bs17580@bristol.ac.uk

Emil Centiu
University of Bristol
zl18810@bristol.ac.uk

Abstract

The existence of a buffer overflow vulnerability in a program allows an attacker to gain control of a system via stack smashing. As a mode of protection against such a vulnerability, the memory of the system is marked to be either writable or executable, but not both at the same time. This paradigm, known as "write-xor-execute" memory, will prevent a malicious user from writing `shellcode` into an executable section of the program. In this paper, we present `ROPOverflow`, a tool that can automatically generate Return-Oriented Programming (ROP) exploits for vulnerable programs.

1 Introduction

This paper describes the mechanism by which `ROPOverflow` works. We assume that the binary we are trying to exploit has a *stack based buffer overflow vulnerability*. A buffer overflow occurs when the volume of data exceeds the capacity of a buffer that the program is writing to. As a result, it will write the remaining bytes of data to adjacent memory locations. Typically, an attacker exploits this issue by directly injecting shellcode and then overwriting the return address with the location of some `shellcode`.

To mitigate this, the "write-xor-execute" paradigm was first introduced in 2003 to protect the memory. In this policy, every page in the address space is either writable or executable, but not both. This is called executable space protection and it is controlled via the `mprotect` system call on Linux systems. Many major CPUs have hardware level support for this, for instance, the XD bit in Intel systems and EVP for AMD devices. Unfortunately, this protection can still be easily bypassed through *Return-Oriented Programming (ROP) exploits*. The concept of *ROP* is simple: a malicious party can chain together jumps to innocuous code in the program, leading to unexpected and potentially dangerous behaviour. These small sequences of instructions are known as *ROP* Gadgets and in most cases end with a `ret` instruction. As these gadgets are found in the executable part of the memory, no code is being run in the writeable part of memory.

Based on the above observations, `ROPOverflow` combines buffer overflows and *Return Oriented Programming* to generate an exploit that can run an arbitrary `shellcode` for any vulnerable binary. `ROPOverflow` is implemented in Python3, and utilises the existing ROPGadget library as well as a pygdbmi API.

There are two seperate versions of the `ROPOverflow` tool: the first (`ropoverflow_execve.py`) was created to exploit the *buffer overflow* vulnerability and insert an `execve` system call with arbitrary parameters whereas the second (`ropoverflow.py`) has been generalised even further to allow an arbitrary `shellcode` exploit.

During development, certain problems have been encountered, for instance the executable binary might not contain all the gadgets needed to perform the exploit or one of the gadgets may be located at a memory address containing a `NULL` byte. This poses a problem as the generated payload must not contain any `NULL` bytes, or else the full payload will not be read. From binary to binary, the locations of gadgets change, so `ROPOverflow` needs to avoid hardcoded addresses. The payload size must also be optimised to be as small as possible as many programs have a limit on how many characters they can read in. The problems and mitigation strategies implemented during the development process of the `ROPOverflow` tool are presented in further detail in Section 3.

Evaluation was performed using separate techniques for the two presented versions of `ROPOverflow`: in order to quantify the performance of the initial stage of the program, a series of command line arguments have been inserted and checked for the validity of the exploit. For the second stage of the program we selected 10 `shellcodes` from the *Exploit Database Shellcodes* by *Offensive Security* for Linux x86 architecture which contains exploit examples. The metrics used to evaluate the performance of the tool were the payload size, the number of `shellcode` exploits successfully executed from the database, the average CPU time it takes to generate the exploit, and gadgets' usage.

2 Background

The “write-xor-execute" memory paradigm was integrated into Linux in order to increase the difficulty of exploiting buffer overflow vulnerabilities for an attacker. Without “write-xor-execute", it is possible to place bytecode instructions directly on the program stack or to alter the address stored in the return register and force the program counter to jump to a malicious payload (possibly preceded by a `NOP`-sled), as presented in [1]. Even though "write-xor-execute" will not allow this type of exploit, it is still possible to alter the control flow by overwriting the return address. A gadget is a piece of machine code that performs a useful function (like writing a value to a register) and ends in a control flow altering instruction (like return). If the executable file is sufficiently large then an attacker is able to carefully craft a chain of gadgets that can get around a non-executable stack. This type of attack is an example of `Return Oriented Programming` and has been introduced in [2]. Return oriented programming has been shown to be resilient to defense strategies that remove functions from the `libc` library or change the assembler's code generation choices.

The exploit demonstrated by Shacham does not only apply to x86 however, it has been shown by Eric Buchanan in [3] that the same idea can be generalised to the SPARC instruction set. The paper covers the design of a high level programming language, the selection of gadgets necessary in order to prove that it is Turing complete and lastly, the design of the compiler that transforms the high level language into gadgets. In H. Shacham's original paper [2] it is also stated that it is possible to create a set of gadgets that is Turing complete for x86 using only libc. Once a Turing complete set of gadgets has been found for a given architecture, then arbitrary computation using those gadgets is possible.

The motivation behind `ROPOverflow` is a desire to have a tool capable of easily creating ROP based exploits for vulnerable 32-bit binary executable files. Without the use of this tool, the process of creating a ROP chain could be challenging. The full process is described later in Section 3.2. Our primary goal therefore was to automate this repetitive and manual process.

Regarding the search for useful gadgets itself, there are multiple different approaches that can be taken. Initially all possible gadgets are gathered without considering if they are usable. This is done by searching the code at the byte level, rather than the instruction level. In the case of x86, it is likely that a given sequence of bytes represents a valid set of instructions as x86 is a dense instruction set (although this may not hold for other instruction sets). The gadget collection step can be performed using the tool `ROPGadget` although other tools exist. After this point, it is necessary to search this set for the gadgets necessary to create a ROP chain. There are two approaches that can be taken, syntax based search or semantics based search. With a syntax based search, the set of gadgets is searched for gadgets that perform a particular instruction or set of instructions. This is simple to implement and is often done using regular expressions. Semantics based search is much harder to implement as the method relies on searching for effects rather than instructions. This is further described in [4] which was an ambitious project to create an all in one tool that could perform a semantic search for gadgets and then chain them together.

A similar tool that uses Return Oriented Programming to inject malicious shellcode into vulnerable programs is `ROPInjector` [5]. First, it reduces the shellcode instructions to their simplest form. After that, it searches for the candidate gadgets in the executable file by their endings. After the gadgets are tested for modification to the registers in question, they are lifted to an intermediate representation. The encoders are responsible to decide whether the instruction can be encoded using their assigned gadget from IR. Some shellcode instructions cannot be converted into ROP using the existing gadgets and in this case `ROPInjector` finds a “`0xCC` cave” where it injects a normal function with standard prologue and epilogue which will be turned into a new gadget. The ROP chain is built on the stack during runtime and each instruction is invoked by a series of stack operations. In order to continue the normal execution flow, the last gadget will return to the instruction that invoked the ROP chain. The `ROPOverflow` tool is different from `ROPInjector` both from an implementation aspect, and from the aim of the tool. `ROPOverflow` targets only buffer overflow exploits and the aim of the tool is to generalize the creation of ROP chain payloads based on arbitrary shellcode exploits.

Another tool that uses a similar principle in order to generate ROP chains from arbitrary shellcodes is `ARG: Automatic ROP Chains Generation` [6]. In this paper, the authors showcase a tool which is able to convert an arbitrary shellcode into a ROP chain without using an Intermediate Language to convert the binary assembly into a meaningful computational representation. As further described in [7], translating low-level machine instructions into a higher level intermediate language represents one of the main steps in binary analysis. Instead, the functionality of the `ARG` tool relies on a system which processes the available gadgets into a Directed Acyclic Graph data structure in order to resolve register access dependencies. The motivation behind the use of a direct translation instead of IL for symbolizing the instructions is that even though direct translation if more difficult to achieve, there are no side-effects due to converting into another language. The `ROPOverflow` tool performs a similar analysis as `ARG` when solving dependencies between register access calls. The `ARG` tool uses the topological sort algorithm to sort the entries in the DAG, our tool performs a brute-force search approach which is further described in Section 3.4.

3 Design & Implementation

The following subsections present the design and implementation choices taken when developing the separate modules of ROPOverflow_execve and ROPOverflow.

3.1 Automatically finding padding length

Overflowing the return address is a crucial part in any buffer overflow exploit. ROPOverflow does this automatically, by running a set of inputs on the vulnerable binary and analysing its behaviour. The test file must not contain any NULL characters (0x00) and, in some cases SPACE characters (0x20) - when the input is passed directly as an argument rather than via a file.

Considering this, the input file contains a consecutive sequence of bytes. For increased efficiency, each byte is repeated 4 times. Therefore, the input file has this structure: 0x01 0x01 0x01 0x01 0x02 0x02... etc. (if the input is passed as an argument it will exclude 0x20 from the sequence). An interesting observation is that this file contains at most 1016 bytes (254×4). Then, ROPOverflow starts running the program with this input inside GDB with Machine Interface. In some cases, GDB will receive a SIGSEGV signal, meaning that the address in EIP register points to a non-executable memory location, and hence RET address, has been altered. GDB provides that altered address, and, if it is a sequence contained in the generated input, it means the RET address has been successfully overwritten. The fact that the input is constructed by a sequence of consecutive bytes, allows ROPOverflow to easily calculate the padding necessary to overflow EIP - just calculate the position in the input of the first byte of the address, accounting for the gap generated by the absence of SPACE (0x20) when necessary.

As mentioned before, this might not be the case for every binary - some will require an input larger than 1016 bytes. In this case, ROPOverflow will generate a new sequence of consecutive bytes and append it to the previous one. The program will be run again and it will append a new sequence until the RET address is successfully overwritten. By doing this, the padding length needs to take into account the length of the previous sequence, giving the correct length no matter how large the input has to be.

While the program is run within GDB, the .data and .bss addresses are also retrieved, by putting a breakpoint inside main and running the command maintenance info sections. These addresses will be used later in the exploit.

3.2 Arbitrary arguments to execve

The ROPOverflow_execve Python program has been created to enable the generalisation of the number of command line arguments given to the execve system call. The program is located in the ropoverflow_execve.py file in the project directory.

The program asks the user to input the command line arguments that will be passed to the execve system call, for example "/bin/sh". The program processes the input in order to ensure that the length of individual arguments is a multiple of 4, so the argument "/bin/sh" will be automatically processed to "/bin//sh". The presented functionality is implemented in the preprocess function.

The two main processes that needed to be generalised in order to accept an arbitrary number of arguments for the execve system call are placing the arguments on the program stack, and creating a shadow stack which will enable storing the start address of the NULL-terminated array of arguments in the ECX register. Placing the arguments on the program stack is performed using the create_stack_ropchain function. The function uses the two gadgets pop eax ; ret and mov dword ptr [edx], eax ; ret in order to place the arguments on the stack.

The other task that has been generalised is placing the memory addresses of the stored command line arguments in a shadow stack. The length of the command line arguments passed to the execve system call including a NULL byte between them plus an arbitrary value of 30 has been chosen as an offset between the address of the start of the stack and the address of the start of the shadow stack. The create_shadow_stack_ropchain function provides the functionality of creating the shadow stack and placing the memory address of the command line arguments on the shadow stack.

The ROP chain ends with resetting the value of the EAX register to 0, incrementing the value 11 times in order to correspond to the value of execve in the system call table, and finally calls the int 0x80 gadget which will lead the program to call the execve system call with the arbitrary given command line arguments.

3.3 Handling addresses that contain NULL bytes

If a NULL byte occurred in the payload then the payload would stop being read by the 'read' system call as it would be interpreted as a NULL terminator. It is therefore imperative to avoid NULL bytes in the payload. The first way to do this is to automatically rule out any gadgets that occur at addresses containing NULL bytes. Whilst it is still possible to create the addresses of these gadgets on the stack (see Section 3.4 of [8]), we decided to just ignore them for the sake of simplicity. Even with this in place however, the values that we need to pop into a register could contain NULL bytes, particularly in the case of the value being an address aligned with a page boundary. To get around this, we considered several different measures.

The first measure that we considered was masking, using either addition, subtraction or XOR depending on the available

gadgets. It is possible to turn any address containing a NULL byte into two separate NULL-free shares, which can be recombined into the original address using one of the previously mentioned gadgets.

This is relatively size efficient, usually taking only 5 gadgets (pop reg1, share1, pop reg2, share2, operation reg1, reg2) and therefore only adding 20 bytes to the payload. This does have the consequence that the contents of reg2 are overwritten and therefore care must be taken regarding the order of operations in the overall ROP chain.

The second measure that we considered was using either 'inc' or 'dec' gadgets to manually adjust the value of an address that is close to the target address. This is a desirable approach for an address that has a NULL byte as it's least significant octet, although it can prove problematic if the NULL byte appears in the most significant octet. If an address only has a NULL byte in it's least significant octet, then address + 1 could be popped from the stack into the register and decremented.

The final measure considered allows a way to avoid the limits of the previous method, however requires a doubling instruction (add reg reg) and an (inc reg) instruction. By using these two operations, it is possible to construct any number bit by bit. This is very effective in the case of small numbers where a masking gadget does not exist, as it requires significantly fewer bytes compared to a pure inc/dec approach.

3.4 Executing arbitrary shellcodes

In order to execute arbitrary shellcodes, we decided to use the 'mprotect' system call on the start of the BSS segment as this location could be determined when we built the payload. The system call is used in order to give read, write and execute permission to a 2 KB region, allowing us to circumvent the "write-xor-execute" restrictions and run the shellcode. As the start address for the mprotect system call has to be page aligned, we call mprotect on (bss & 0xfffff000).

We chose to simplify moving the shellcode to the BSS segment as well as shorten the length of the payload by using a 'read' system call in order to take the shellcode directly from stdin into the BSS section. This means we can avoid using the write-what-where gadgets and serves to reduce the complexity of the ROP chain, as well as it's length. The shellcode is read into the offset of (bss & 0xfffff000) + 4 in order to avoid needing to jump to an address containing a NULL byte.

In order to generalise the ROP chain, we created a method that allows a high level approach to be taken, focusing on the values in each register rather than the gadgets themselves. This is closer to a semantic search rather than syntactic search, because we search for gadgets that have some effect, rather than searching for specific gadgets.

For each register, ROPOverflow searches for gadgets that are necessary to create the ROP chain, including any gadgets to handle dealing with a NULL byte if required. Every possible gadget sequence that stores the desired value in the register is then stored in a set, sorted from shortest to longest as well as has their dependencies tracked. In this case, dependencies are the registers that will be overwritten by the sequence. After this, the sets of sequences for each register are ran through a scheduling function that returns an ordering of the gadgets which respects the dependencies (no register is overwritten after it has been written to) and that also minimises the length. Even though more complex algorithms exist, this is done using a brute-force algorithm because the problem size is small enough that the runtime is dominated by the other sections.

4 Evaluation

4.1 ROPOverflow_execve

The performance of the ROPOverflow_execve tool has been evaluated in terms of time taken to process the ROP chain exploit file, number of commands that can be processed, and number of vulnerable binary files that can be exploited using the output from the program.

The platform used to perform the evaluation of the ropoverflow_execve.py program is a Ubuntu 18.04.5 Bionic Beaver machine with a Intel(R) Core(TM) i7-7500U 4-core CPU and 16GB of RAM. The average time taken by the ROPOverflow_execve tool in order to generate the file containing the ROP chain is 1.3 seconds. In order to calculate the average runtime of the program, 20 vulnerable binaries have been exploited. The source code used to generate the binaries vulnerable to stack buffer overflow exploit is available in the vulnerable_programs/ subdirectory of the project.

In order to demonstrate that the ROPOverflow_execve program is able to successfully process arbitrary command line arguments, the following commands have been passed as input to the program: /bin/sh, /bin//sh, /tmp/nc -lnp 5678 -tte /bin/sh, /tmp//nc -lnp 5678 -tte /bin//sh, /bin/sh ./exe/crl.sh, /bin/sh ./exe/lyn.sh. These commands showcase the ability of the program to successfully process arbitrary command line arguments for the execve system call. The scripts available in the exe/ subdirectory are scripts that run arbitrary Linux command line utilities, such as curl and lynx. The scripts showcase the ability of the program to create a ROP chain that can use command line utilities to create, delete or download files, which can be used to download a malicious program and execute it, or upload one on an accessible server.

4.2 ROPOverflow

ROPOverflow was evaluated in terms of payload size (the ROP chain exploit), the time taken to generate the exploit, gadgets' availability, gadgets' usage, and also the success rate.

The binary used for testing the payload size was vuln3-32-test. For all 10 shellcodes, the payload size stayed consistent at 240 bytes (including the junk data). This

is because ROPOverflow passes the shellcode via a read system call which is set to read 0xffffffff bytes which means that it will read until the End of File (EoF)

The evaluation platform of ROPOverflow.py is Ubuntu 18.04.5 Bionic Beaver (64-bit) machine with a Intel(R) Core(TM) i5-7300HQ @ 2.5GHz x 4 CPU. The total time it took to generate the exploit was approximately 3.5 seconds. This is split between finding the necessary junk data length to overflow the RET address, and finding the gadgets in the vulnerable program and constructing the ROP chain that will execute the shellcode (each is approximately 1.2-1.3 seconds).

The success rate was measured by running all 10 shellcodes (add-root-user, shredtest, adduser, breakchroot, chroot, flush-ip-tables, killall, setreuid_chroot-reak_execve_bin_sh, shred-file-test, shellcode) within vuln3-32-test. They all succeeded in doing what was expected, hence the success rate is 100%.

Gadgets' usage is consistent throughout the test binaries, ranging between 34 to 36. Although it is more likely to find more gadgets if the binary size is higher, the number of the gadgets found is not dependent on it. The binaries used were netperf 2.6.0, sipp-3.3, vuln3-32-test as well as the previously mentioned binaries used to test ROPOverflow_execve.

4.3 Limitations of our work

Programs can receive their input in many different ways, hence the function for finding the padding length is limited to a small set of input procedures. Currently it supports input via file and input as an argument, but it is unlikely that this will work for every binary. Also, it is possible, but unlikely, that when finding the padding length, the altered RET address will point to an executable part of the memory, and thus the program will nor SEGFAULT. This could be mitigated somewhat by using a small subset of the bytes we are currently using to find the padding length.

Because of the way the program is coded, there are still restrictions in the types of gadgets required for it to successfully produce a payload. For instance, if the program is unable to find a gadget which can pop a value from the stack for a given register, it *may* be unable to generate the payload successfully. Although we have implemented zero and inc as well as double and inc in order to push a value to the register, there are still many other possible ways of doing it that we have not considered. Regardless of what sequences are searched for by our program, it is impossible for us to have implemented an exhaustive list. Fortunately, ROPOverflow can be easily expanded to support the search of further sequences, which could allow it to discover new payloads where it was previously unable, or to discover shorter versions of existing payloads.

There is another limitation regarding the shellcodes that can be run by our program. Firstly, the shellcodes that it can run must be NULL free. This is because they are read in using a read system call from stdin, meaning any NULL bytes will stop the file from being read. This could be mitigated by instead reading the shellcode directly from a file but that would come at the cost of payload size and (potentially) requiring more gadgets. Any shellcode run by the program cannot assume the value of any register when it starts. The values are not guaranteed to be any particular value. Again, this limitation could potentially be worked around by explicitly adding code to set the value of the registers before the shellcode starts, but this would once again increase the size of the payload or the gadget requirements. The shellcode must not assume the region that it is inserted into is zeroed, as this is not necessarily the case. This can be a problem if a shellcode ends with a string that is not NULL terminated. If the surrounding area was zeroed this would not be a problem, but because it is not this can lead to strings being extended by reading nearby memory and shellcode failing.

One final limitation lies in the assumptions we make about the vulnerable binary. For the purposes of our testing, we compiled the program with the -static and -fno-stack-protector flags which makes performing the exploits significantly easier. Because -static was enabled, the binary contained all of the gadgets from the imported libraries (including libc) which significantly increases the ease of locating and using gadgets. If the static flag was omitted then the library would be dynamically linked at runtime and the offsets would no longer necessarily be predictable. The paper "*ROP Gadgets Hiding Techniques In Open Source Projects*" [9] describes a potential method to ensure the existence of necessary gadgets in open source code by making a seemingly innocuous commit that can add useful gadgets to the binary once it has been compiled although it may still be possible to find gadgets in dynamically linked code regardless. If the -fno-stack-protector flag was omitted, then the program would be compiled with stack canaries. This means that before a function attempts to return, it checks a value stored below the return address on the stack. If the value has been modified, then the program terminates before the function returns. This causes a far greater problem for performing our attack, as unless our payload was able to guess the canary value (which is extremely unlikely) or obtain it through some other means, we would be unable to perform any ROP exploit.

5 Conclusion

This paper has showcased the two Python tools ROPOverflow and ROPOverflow_execve and their corresponding limitations. Following evaluation, the developed programs have demonstrated that they can achieve the predefined requirements and that they provide a more generalized approach to generating Return Oriented Programming exploits.

References

[1] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), 1996. http://phrack.org/issues/49/14.html.

[2] Hovav Shacham. The geometry of innocent flesh on the bone:return-into-libc without function calls (on the x86). In *Proceedings of ACM CCS 2007, ACM Press*, 2007. https://hovav.net/ucsd/dist/geometry.pdf.

[3] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, page 27–38, New York, NY, USA, 2008. Association for Computing Machinery.

[4] NGUYEN Anh Quynh. Optirop: Hunting for rop gadgets in style. 2013. https://media.blackhat.com/us-13/.

[5] C. Ntantogian, G. Poulios, G. Karopoulos, and C. Xenakis. Transforming malicious code to rop gadgets for antivirus evasion. *IET Information Security*, 13(6):570–578, 2019. https://ieeexplore.ieee.org/abstract/document/8890330.

[6] Y. Wei, S. Luo, J. Zhuge, J. Gao, E. Zheng, B. Li, and L. Pan. Arg: Automatic rop chains generation. *IEEE Access*, 7:120152–120163, 2019.

[7] Niranjan Hasabnis and R. Sekar. Lifting assembly to intermediate representation: A novel approach leveraging compilers. *SIGPLAN Not.*, 51(4):311–324, March 2016.

[8] Nergal. Advanced return-into-lib(c) exploits (pax case study). *Phrack Magazine*, 11(58), 2001. http://phrack.org/issues/58/4.html.

[9] M. Prandini, M. Ramilli, and M. Prati. Rop gadgets hiding techniques in open source projects. 2012.

A Group Member Score

- Ruairi Fox (rf17160) - 33%
- Bogdan Stelea (bs17580) - 33%
- Emil Centiu (zl18810) - 33%

B Individual Contributions

B.1 Ruairi Fox (rf17160)

Initially, I was responsible for making sure that `ropoverflow_execve.py` was able to handle addresses that contained `NULL` bytes. After that, I took the lead on the collection of gadgets, as well as creating a general system to store them and then build a ROP chain for `ropoverflow.py`.

For the report, I wrote sections 3.3, 3.4 and 4.3 as well as parts of the introduction and the background. I also recorded the demo of `ropoverflow.py`.

B.2 Bogdan Stelea (bs17580)

In the beginning of the project, I have been responsible for the creation of the module generalizing the number of command line arguments for the `execve` system call, which later became the `ropoverflow_execve.py` program. Also, I have created the `Vagrantfile` used to deploy a Virtual Machine running Ubuntu, which allowed us to develop on the same environment. After the `ROPOverflow_execve` program responsible with generating a ROP chain file given arbitrary command line arguments for the `execve` system call was successfully implemented, I have done literature review, reading academic papers and blogs in order to gain a deeper understanding of the task described at the fourth point of the project, namely using Return Oriented Programming to allow any arbitrary shellcode to be executed. I was tasked to write the *Abstract* and *Background* sections of the paper, along with the *Arbitrary arguments to execve* subsection of the *Design & Implementation* section of the paper, as well as the *ROPOverflow_execve* subsection of the *Evaluation* section of the paper.

While performing the evaluation of the `ropoverflow_execve.py` program, I have created 20 further vulnerable C programs available in the `vulnerable_programs/` subdirectory of the project. Futhermore, I have created the shell scripts executed during the evaluation of the `ROPOverflow_execve` tool available in the `exe/` subdirectory of the project. I have also performed the evaluation task for the `ropoverflow_execve.py` program.

I have also participated during pair programming integration meetings with my teammates which were held every 2 days during the coursework assessment period. Furthermore, I also took part in the creation of the video presentation of the project.

B.3 Emil Centiu (zl18810)

I was initially responsible for creating the module `input_length.py` which finds the junk data length and also the `.data` and `.bss` addresses. I also built the module `exploit_gadgets.py` which returns the ROPgadgets of the binary. After that, apart from small bugfixes and tweaks, I focused on the evaluation part, building the `eval/eval.py` module, compiling known-vulnerable programs, adding shellcodes (alongside Ruari).

As for the report, I wrote sections *Introduction*, *3.1 Automatically finding the padding length* and 4.2 *ROPOverflow*

Evaluation.

Finally, I participated in all pair programming sessions with my teammates, and recorded the evaluation part of the video presentation.

C Project Proposal Requirements Achieved

The project documented in this paper has been realised by following the requirements specified at **Project 4** available on the COMSM0051: Systems & Software Security Coursework Unit page. In the initial Project Proposal we have defined the following project requirements:

- Generate a string with a pattern (e.g. `0x01 0x02 0x03 0x04 ...`) and by observing where the program attempts to return, detect the length of padding needed
- Generalise the arguments passed to the execve ROP-chain created by ROPGadget
- Calculate an address that needs a `NULL` byte in the registers, rather than read it from the stack. We will test the implementation by attempting to create shadow stack at a memory location containing \x00
- Create a ROP-chain to call the `mprotect` system call and make the stack executable, then run the shellcode on the stack.

We have managed to implement all of the mentioned requirements, with the `ROPOverflow_execve` tool satisfying the first three requirements, and the `ROPOverflow` tool satisfying all of the presented requirements. We have managed to split the tasks of implementation, literature review and evaluation between us and integrated the developed modules successfully.

An alternative implementation strategy for the final task of our project could have been converting the shellcode into an Intermediate Representation in order to later be transformed into a chain of ROP gadgets, or by implementing a similar approach as the one taken in the paper *ARG: Automatic ROP Chains Generation* [6]. We have managed to implement in our solution a similar approach to the one in the mentioned paper regarding finding and processing the ROP gadgets by filtering their dependencies, and then using a brute force approach to find the optimal solution. Due to the time constraint and the complexity of the task, we have decided to implement the solution as mentioned at the last point in the project requirements list which implied using the `mprotect` system call to alter the stack protection to be executable and then calling the `read` system call to pass the arbitrary shellcode to be executed by the vulnerable program.

D Github Repository

The project source code can be found in the following private Github repository: `https://github.com/bstelea/s3`.