SSS Coursework Report

Gabriel Galadyk
School of Computer Science
University of Bristol
us21186@bristol.ac.uk

Deepkumar Pawar
School of Computer Science
University of Bristol
qz21635@bristol.ac.uk

I. Introduction

Automated exploit generation (AEG) is the process of automatically identifying vulnerabilities in a program and generating the appropriate exploit that abuses the vulnerability found, usually this exploit involves launching shellcode on the target system.

Our project deals with a specific subset of vulnerabilities that involve exploiting the stack to achieve shellcode execution on arbitrary C programs; we achieve this usually with a buffer overflow exploit that permits us to overwrite the return address of the function to any arbitrary address which leads to executing either shellcode or a ROP chain (depending on mitigations) that leads to the launching of a shell.

Additionally, our project deals with scenarios where certain mitigations are enabled on the system, such mitigations include enabled ASLR and disabled stack execution which further expands the class of programs that our project is capable of exploiting.

II. Background

With the growth of the internet around the world and the rise in use of devices, the potential for malicious actors finding ways to exploit and break into systems grows larger. It is estimated that hacking has cost the global economy over $450 billion annually[1] and for many keeping their devices secure is the most important design consideration.

The act of breaking into a system or hacking into a computer usually necessitates the discovery of a vulnerability or an exploit in the software or hardware of the program, such exploits usually only exist due to human error or oversight in the creation of the hardware[2] or during the creation of the software[3]. As a consequence, the field of cybersecurity and research into protecting programs, or otherwise identifying vulnerabilities to patch them before bad actors discover them first exists in order to satisfy the need to protect programs and protect the data they store/systems they are on.

One of the results of this field of thinking was AEG for the purpose of automating the act of discovering vulnerabilities, and generating the exploits to abuse them so that the software developers are able to effectively and quickly patch the potential exploits and better protect their programs.[4]

AEG has quickly exploded as a primary method for analysing programs for potential vulnerabilities within software and has led to the creation of many sophisticated AEG tools like Mayhem[5] which was at the time capable enough to find 2 previously undiscovered vulnerabilities within the Windows and Linux operating systems.

III. Design & Implementation

Our program assumes that within our target exists a potential stack overflow vulnerability, most often than not in the form of a buffer overflow exploit which is considered to be one of the most common security vulnerabilities in software[6]. We have chosen to utilise the GDB debugger (for finding input string length sufficient to overwrite the return address) due to its ability to be automated with the usage of a Python script which interacts with its API.

The entirety of our implementation including all the features discussed below are bundled into a single main python file which when ran with flags as described in the READ.ME file, lead to automated finding of input string length sufficient to overwrite the return address and either of two ROP based exploit generation.

A. Fuzzing

Assuming the user has not already provided information about the offset required to overwrite the return address, our program performs a basic fuzz for potential buffer overflow exploits. It generates a cyclic pattern and creates a file input and a string input to run within GDB. One of the major restrictions of this is due to the fact that the generation of the cyclic pattern was implemented directly without the usage of a library, great difficulty was experienced in integrating python modules to use in the script due to difference in environment, and the result is a slow albeit sufficient cyclic pattern generator and cyclic pattern finder. The choice of writing our own fuzzer component stemmed primarily from the fact that utilising existing fuzzers was unfeasable due to issues in operating them.

First, we attempt to run the program in GDB using command line input, this will attempt to potentially find buffer overflow exploits on functions like scanf and gets and see if the return address has been overwritten by checking if the program has had a segmentation fault. If the program has had

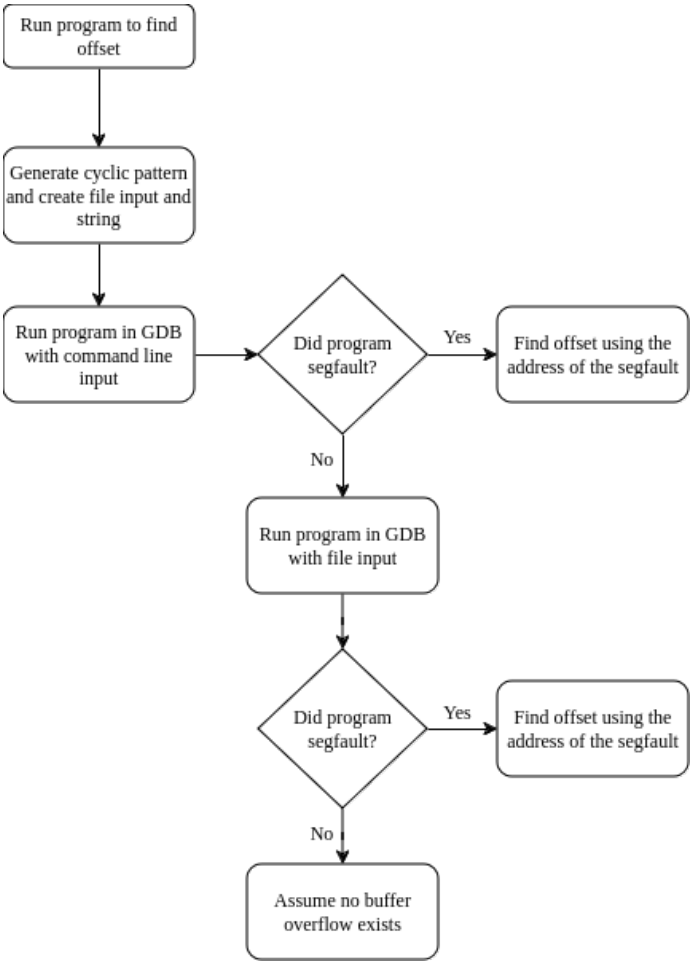

Fig. 1: Flowchart of the fuzzer component of the program.

a segmentation fault, the script then attempts to locate how big of an offset it requires in order to overwrite the return address and provides that value to the rest of the program to guide the actual exploit generation.

Supposing that the script did not find a segfault, then the program will rerun using file input, this will attempt to find potential buffer overflows in functions like ‘fgets‘ and ‘fread‘. Again, if the program does encounter a segmentation fault, then it attempts to find the offset required to overwrite the return address and as above it provides that value to the rest of the program.

In the case that there isn't any segfault, then the script would have preferably attempted a format string exploit to identify the stack overflow exploit.

B. ROP Chain for Execve Syscall

Given a binary file with a buffer overflow vulnerability, an arbitrary command to execve and the input string length sufficient to exploit the vulnerability to overwrite the return address (taken from our fuzzer implementation above or otherwise), we present a program that is able to generate a ROP chain to exploit the vulnerability and call the execve systemcall to execute the given command. For this implementation we rely on being given precomputed gadgets for a given binary file using tools such as ROPgadget **CITE**. Our implementation consists of the following sequential steps.

1) Get Gadgets: We predetermine the gadgets we need and find them in the output given by the ROPgadget tool. If these gadgets are not found, the exploit generation will fail. However, for statically linked files, the gadget selection produced by ROPgadget largely stays the same and so practically this is not an issue. The reason for this is because ROPgadget finds gadgets by disassembling the the binary file using capstone to look for gadgets. Since most programs will use library functions from libc for example, in such a statically linked program, the binary will contain all the code required for it to execute including the code for these popular library functions where most of the gadgets found by ROPgadget reside. Therefore, the selection of gadgets remains largely the same and the exploit works well. We evaluate further in section IV.

2) Get .data Address: To pass arguments for the execve syscall, the calling convention for Intel x86 requires us to build place the arguments in memory with specific registers pointing to them. We decided to use the memory at the .data header address of the binary file. This is for two reasons. Firstly, it is on most Intel x86 systems a readable as well as writeable part of memory, which is what we need to carry this exploit out successfully and is not the case for many other parts of memory. Secondly, it was easy to automate finding the address of this section which we did by examining the objdump of the given binary file and finding the required address in it.

3) Construct ROP Chain: We construct the ROP chain by first appending the padding required to start overwriting the return address. Then construct the ROP chain in two distinct stages. The first involves using POP and MOV gadgets to place the command line arguments into the .data address sequentially, separated by null bytes as required. This posed an initial struggle since we could not use null bytes in our ROP chain as a strcpy (more common means of a buffer overflow vulnerability) would stop copying once it encounters a null byte. We proceeded by moving a zeroed out $EAX register using an XOR instruction. The second stage involved placing the required values into the $EBX, $ECX and $EDX registers as per the calling convention of execve as well as changing $EAX to the right value for the execve system call. Our ROP chain is now ready and is written to a file.

C. Arbitrary Shellcode Execution using ROP

To achieve arbitrary shellcode execution using a ROP based exploit, we follow a similar set of steps as discussed above

with the key difference being the construction of the ROP chain. The idea was to move the shellcode to a location in memory and then 'return' there to execute the shellcode as if they were instructions somewhere in the program's virtual memory address space. The key struggle however, is to find a memory address which is marked as both writeable and executable. Such memory locations are difficult to find directly, especially amongst the sections in a binary file such as the .data section, usually as a protection mechanism against exploits.

There are two system calls that are useful in this case, the mprotect syscall and the mmap system call. We use the mprotect system call which allows us to modify the access permissions of a supplied memory address. We could have also used the mmap system call which directly finds us a memory address with requested access permission and enough given length. We evaluate them in section IV.

Constructing the ROP chain to prepare the arguments for the mprotect system call was challenging due to the requirement of the memory address being page aligned, i.e. being a multiple of the page length for the given architecture and system, which in our case for Intel x86 was 4096 bytes. Getting these values into the right registers whilst avoiding null bytes needed an elaborate combination of available gadgets whilst taking care of potential side effects (gadgets that change the values of unintended registers).

We chose to use the .data memory address again for this. Usually .data is writeable but not executable but this posed no issue when using the mprotect system call. Then we loaded the shellcode into the address, with special consideration for any potential null bytes. Finally we put the .data address itself at the end of the ROP chain for the EIP to 'return' to where placed our malicious shellcode, ready to execute.

IV. Evaluation

We evaluate our AEG by analysing the subset of programs it works on and where it fails followed by a brief discussion on improvements that could help address these issues. The fuzzer component will also be compared to other fuzzers that exist by comparing their capabilities respectively.

A. Fuzzer

	Our Fuzzer	AFL	Hongfuzz
Input Probing	Has	Has	Has
Black-Boxing	Has	Lacks	Has
Specific Format Input	Lacks	Has	Has
Custom Files Input	Lacks	Has	Has
Mutation	Lacks	Has	Has

Table 1: Features Comparison of fuzzers

Our fuzzer is a very simple script that interacts with a given program by providing it intentionally malicious input with the assumption that a buffer overflow exploit might exist within the code, for simple programs that do not require very specific input (XML/JSON files, audio files, etc) the fuzzer achieves its goals with the desired effect of obtaining an offset, but on more complicated programs (multiple inputs/specific input formats) the fuzzer would experience difficulty or fail, in addition, security mechanisms that make stack smashing harder like stack canaries would defeat the fuzzer. The fuzzer is capable of theoretically finding any buffer overflow exploit, however because it does not discriminate between vulnerable buffers and non-vulnerable buffers, it might produce results for buffers that are not vulnerable and additionally it might produce an incorrect when a part of the initial pattern gets placed into a seperate non-vulnerable buffer while the remainder infiltrates the vulnerable buffer, as a result, the fuzzer becomes weaker as the program grows larger and has more and more buffers.

Table 1 shows the general lack of features the fuzzer has compared to AFL and Hongfuzz fuzzers, our fuzzer can claim to do black-boxing by virtue of the fact that our fuzzer does not analyse the program's source code in any way and focuses on seeing if a segmentation fault was caused.

B. ROP Chain for Execve Syscall

Our exploit generation only works for statically linked programs. This is because the ROPgadget tool fails to find enough gadgets for our exploit generation to build the required ROP chain when used on dynamically linked programs. This is because ROPgadget uses capstone to find the required gadgets in the binary file itself. In a dynamically linked file, where the addresses to various library functions and thus libc are not present before runtime, the tool has a much smaller space to find gadgets in.

Secondly, our exploit generation only works when Address Space Layout Randomisation (ASLR) is turned off. This is because the address to .data as well as the absolute addresses to the various gadgets are found before runtime and if ASLR was turned on, these addresses would change, leading to our ROP chain failing.

The selection of gadgets used in the exploit generation remains constant and relies on those exact gadgets being found by the ROPgadget tool. However, we argue that this does not affect our implementation for statically linked programs as most if not all of our gadgets used are found in libc, which is commonly called at least once by most of the programs vulnerable to a buffer overflow exploit (strcpy, etc.).

See Table 2 which illustrates our point. Statically linked programs with only a couple library function calls contain thousands of gadgets. We empirically confirm that the gadgets found for the statically linked programs in the table contain

all the gadgets we require for our exploit generation to work successfully. However, the dynamically linked programs have significantly less gadgets we empirically confirm that they do not contain the gadgets necessary for our exploit generation to work successfully.

Example Program[Linkage]	Library Functions	Gadgets
example1.c[static]	strcpy, printf, fprintf, frread	37445
example1.c[dynamic]	As above	231
example2.c[static]	fopen, fread	37541
example3.c[static]	strcpy, strcat, printf	43537
example3.c[dynamic]	as above	153
example4.c[static]	printf, scanf, sprintf	42719

Table 2: Gadgets found for different C files

C. Arbitrary Shellcode Execution using ROP

We evaluate our method of ROP exploit generation for executing arbitrary shellcode in two respects.

Firstly we compare the usage of the mprotect and mmap system calls to obtain an address in memory that is executable to copy the shellcode into and execute. In our implementation, we chose to use mprotect instead of mmap. We start by noting that using an mmap system call is slightly easier as the calling convention is easier to satisfy. The mmap system call gives us the address of a suitable range in memory with the required access permissions. We do not have to deal with page alignment of the memory address at all. The only tricky part comes in trying take the given memory address and push it into the stack as needed. When using mprotect, we require an elaborate sequence of gadgets to get the aligned memory address into the right register along with an aligned length. These values contain a lot of null bytes which make it difficult to construct. However, using mprotect is more effective than using mmap. This is because mmap tries to find addresses in memory with the supplied length and access permissions. There is no guarantee that such a memory address would be found, particularly for large shellcodes. Using mprotect allows us to be much more resilient to such limitations. Thus, an implementation with mprotect, such as ours, is better than an implementation using mmap.

Secondly, we compare our method of executing shellcode against the method used in [7]. The key difference is that their method involves the direct translation of shellcode into a ROP chain whereas our method simply uses mprotect to get an executable part of memory and pastes the shellcode there ready to be executed. The benefit of our method is that it leads to a much smaller payload since it contains the shellcode as it is, with a constant overhead of the loading into the memory address ROP instructions. This is compared to their method, where the direct translation of shellcode to a ROP chain would likely lead to a larger variable overhead of multiple return addresses to execute a single shellcode instruction. It could be argued that this difference in overhead is mitigated by their method of injecting custom gadgets as required in memory caves, and depending on the shellcode, it might even be less than the length of our ROP chain due to our constant overhead mentioned earlier.

However, we then mention that they have to adhere to a subset of translatable instructions that they have accounted for and can be converted to a ROP chain. Notably, this is not feasible for branch instructions which they have not implemented since for a ROP chain to work, there must be a return address to facilitate returning back to where the gadget was called from to call the next gadget to continue our ROP chain (it might still be theoretically possible to chain branch instructions together and get to a return, but in the likely case this is extremely difficult or unfeasible). Our implementation is not limited in this way and works as long as the shellcode works.

We must note however that the purpose of the implementation described in this paper is for antivirus evasion, which it succeeds at. Our implementation would be poor at antivirus evasion for at least three reasons. Most antiviruses are adept at picking up on shellcode that exists in memory, due to it's direct and abnormal placement in memory. Also, calling mprotect to change the access permissions of a memory address to writeable and executable tends to be flagged by antiviruses. Finally, our implementation is a runtime exploit whereas their implementation has the ability to infect a binary and avoid antivirus detection.

V. Conclusion

We conclude by reiterating on our achievements in this paper.

We have created a simple fuzzer that is theoretically capable of finding any buffer overflow exploit without the need for the source code of the program and is capable of handling programs with file inputs and with user inputs.

Furthermore, we are able to automatically generate ROP based exploits that can execve arbitrary command line as well as execute an arbitrary shellcode for statically linked programs when ASLR is turned off.

For future work, we would want to find a way to get gadgets from dynamically linked programs. We believe this is possible using a GOT overwrite type exploit which would exploit a format string vulnerability. By exploiting this vulnerability it might be possible to get the address of the library functions and where they are stored in memory and using a disassembler, such as capstone or otherwise, we would have access to a larger space of instructions to find gadgets in. It would also be very interesting to discuss the limitations or difficulties with this type of exploit as we encounter Position Independent Executable or PIE and how it differs with ASLR and it being a

measure applied at compile time when ASLR is based purely on the run time environment.

APPENDIX

We met our base objectives well, as per the specification given in the project brief, achieving all the mentioned points. We went above and beyond our initial targets by writing a simple fuzzer that is able to theoretically account for all types of buffer overflow vulnerabilities.

Unfortunately, we were unable to implement a planned GOT overwrite type exploit as mentioned in the conclusion section of the paper. This was a bit disappointing as we had gotten to the stage of understanding and being to execute a GOT overwrite exploit, but were ultimately unable to implement an automation of the same due to time constraints, and so could sufficiently explore how or if this might help us find gadgets for a dynamically linked program.

It was very difficult to figure out an elaborate sequence of gadgets for the ROP chain that calls mprotect and struggles with null bytes. However, once it was figured out, it did not cause an issue between different binary files (statically linked, of course).

Additionally, we were unable to fully implement a comprehensive fuzzer, there were many difficulties with investigating the feasability and what sort of fuzzer we would need, many of these difficulties came from terrible source repository instructions and broken features that hampered research into the fuzzer, there were also additional difficulties with using Python and dealing with the various Conda and Mamba environments that built up over the years

Individual contributions:

Deep worked on the execve and shellcode ROP exploit generation. Gabriel worked on the fuzzer implementation. We worked on approximately 50% of the report each. We agree to a 60% and 40% overall contribution split with Deep contributing 60% and Gabriel contributing 40%.

REFERENCES

[1] S. Samtani, Y. Chai, and H. Chen, "Linking exploits from the dark web to known vulnerabilities for proactive cyber threat intelligence: An attention-based deep structured semantic model," *MIS quarterly*, vol. 46, no. 2, 2022.

[2] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2020. DOI: 10.1109/TCAD.2019.2915318.

[3] D. Everson, L. Cheng, and Z. Zhang, "Log4shell: Redefining the web attack surface," in *Proc. Workshop Meas., Attacks, Defenses Web (MADWeb)*, 2022, pp. 1–8.

[4] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Commun. ACM*, vol. 57, no. 2, pp. 74–84, Feb. 2014, ISSN: 0001-0782. DOI: 10.1145/2560217.2560219. [Online]. Available: https://doi.org/10.1145/2560217.2560219.

[5] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*, IEEE, 2012, pp. 380–394.

[6] K.-S. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Software: Practice and Experience*, vol. 33, no. 5, pp. 423–460, 2003. DOI: https://doi.org/10.1002/spe.515. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.515. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.515.

[7] C. Ntantogian, G. Poulios, G. Karopoulos, and C. Xenakis, "Transforming malicious code to rop gadgets for antivirus evasion," *IET Information Security*, vol. 13, no. 6, pp. 570–578, 2019.