A Tool that Segfaults

1 Introduction

This project aims to develop a tool which will, given the path of a binary and an arbitrary bash command, exploit said binary to execute this command via execve by making use of ROP gadgets. Use of the tool will assume an x86 architecture, and that the binary has been compiled with the static flag, i.e. does not rely on dynamically linked libraries, and that the stack protector is turned off. It will also rely on ASLR not being activated.

2 Background

Return-Oriented Programming (ROP) is an exploitation technique, used particularly in attacks buffer overflow vulnerabilities. First developed as a method to circumvent the deployment of executable-space protection, where regions of memory are marked as non-executable, making use of the NX bit, to stop machine code from being executed in these regions. ROP allows attackers to construct a payload for existing code in the system, code which already has permission to be executed. It works by carefully chaining small machine instruction sequences that already exist in executable address space, known as "gadgets"; these typically end in a return instruction. By using gadgets to manipulate a program's control flow, attackers can execute arbitrary commands allowing full control over the computer. ROP attacks have become frequent in modern development of exploits, especially when it comes to defeating Data Execution Prevention (DEP) and Address Space Layout randomisation (ASLR).

ROP is a type of stack smashing, it generally relies on the manipulation of a call stack by taking advantage of a poorly written or buggy program, this is a buffer overflow most of the time. Occurring when a program doesn't perform proper bound checking when storing data provided by a user into its memory. This allows for more data to be stored than it is prepared for, therefore the data may overwrite space allocated to function variables on the stack and rewrite their return addresses. These overwritten addresses can be redirected to a location of the attacker's choice, in this instance, the start of a ROP chain.

3 Design & Implementation

3.1 Finding the Overflow

In order to find the buffer length needed to overflow a program, we employed use of a binary search. The program is compiled statically without use of the '-g' flag. The first step is to find the addresses that we want to put breakpoints at, ideally the last assembly instruction before the return of each function. To do this, `objdump -d binary > disassembly.asm` is used to generate an assembly file of the binary we want to exploit, and pipes it into a file. Then we can parse the file to find the functions in between the `frame_dummy` and `main` functions, and extract the instruction address that precedes the `ret` instruction. At this point the binary search is initiated, starting at an arbitrary buffer length, breakpoints are set at the identified addresses. The binary is run with an input file containing the buffer length of 'A's piped into it, and the contents of the `ebp` register and the following 12 bytes are read to determine whether an overflow has occurred. The length of the buffer is adjusted iteratively through a binary search so that the optimal length of buffer is found; when only the `ebp` register is overflown and nothing more. Once the buffer length is determined, it is piped back into the main ROP chain attack implementation. This implementation can find the buffer length from programs with varying buffer lengths, implementations of functions, as well as different numbers of functions.

3.2 Defeating ASLR

In the defeat of ASLR, we based our attack on the ret-to-libc. In order to implement this attack, the program has to be compiled dynamically with the `-g` flag. Following this, we use `objdump -d` to generate an assembly `.asm` file like for

the previous method. However, the different compilation options mean that the file was in a different format and needed a unique way to parse it to find the functions of interest. For the breakpoints, both the start and end of the instructions of the functions are needed as breakpoints are defined in the form `b *copyData+0x2d` where `0x2d` is the size of the function minus one. Using these breakpoints, we found the buffer length the same as above.

To bypass the effect of ALSR, an address for the `libc` library is found using `ldd binary | grep libc`, and then using `pygdbmi` and the command `p exit` the exit address can be found, allowing us to exit gracefully after our attack has been implemented. On top of this, the location of `/bin/sh` is found within `libc` These addresses are added to the input file after the buffer length of bytes. ASLR introduces randomisation of 2^8 locations of `libc` in 32 bit architecture; a relatively small number of attempts are necessary for a successful attack. Therefore, by running the binary with the input in an infinite loop, the address of `libc` in the input file and the address of `libc` in the memory will match, hence triggering the command `system(/bin/sh)` and gracefully exiting. This method only works efficiently on 32 bit architecture due to the smaller number of attempts that are needed to be made, an attack like this wouldn't be feasible on a 64 bit binary.

3.3 Gathering the Gadgets

We made use of ROPgadget to find all individual gadgets available to the given binary, piping the resulting output stream over to our own Gadget parser. This stream of gadgets would be supplied along with a list of INSTRUCTION objects, consisting of the assembly instruction as a string literal to search for and the registers to be preserved before the required instruction is executed, as well as the registers to be preserved after. This helps our parser decide which gadgets to disregard when looking for ones to fulfil the desired instruction. For example, if we are searching for POP EAX, we wouldn't need EAX to be preserved before the POP is executed, but we would need it to be preserved after. i.e. the gadget MOV EAX,EDI ; POP EAX ; INC EBX ; RET would be accepted, whilst the gadget ADD EAX, ECX ; POP EAX ; MOV EAX, DWORD PTR [EDX] would be rejected. Register preservation requirements can usually be inferred from the nature of the instruction, but this becomes more interesting when considering the requirements in the context of the wider program - this is discussed further in **3.3**.

It is also required that the gadgets end in RET instruciton, to return control back to our ROPchain in the stack. It is possible to also include gadgets which end in a JMP {ADDR} (if the address points to another gadget or attacker controlled part of memory) or ending in POP {REG} ; JMP {REG} [1]. In either case, we can effectively chain gadgets together as explored in [2]. This allows the formation of more complex gadgets; ones which could also be harder to be identified by ROP countermeasures, but would require more manipulation to build into a desired sequence. Due to the latter, we aim for less complex gadgets. Once a gadget is accepted, its complexity is compared with the existing gadget (if any) corresponding to the required instruction. Complexity in this case is simply the number of instructions which make up a gadget. Finding ones which are shorter in length reduces the risk of unwanted register modification. Out gadget finder ultimately finds the least complex gadgets it can - as close as possible to just INSTRUCTION ; RET.

., At the end of the parsing process, our GadgetFinder class will have a set of gadgets, each consisting of the string literal of the original instruction the gadget is used to emulate, the address of the gadget, its complexity score and a list of 'side pops'. These are extraneous registers which are popped into during execution of the gadget. e.g. the gadget POP ECX -> POP EXC ; POP EBX ; RET would have the EBX being side popped. This is effectively a more limited version of the Gadget Summaries used in [3] which summarises the pre and post side affects. When building a program out of these ROP gadgets, tracking these side-effects is useful in determining whether we need additional gadgets to preserve correctness. This could be extended further by accounting for MOV side-effects, as well as those of arithmetic operations of form {OP} {DST}, {SRC}.

3.4 Arbitrary Execve

Once the amount of padding sufficient to overwrite the return address is found, we can replace this address with the start of our ROP chain. The aim of our chain will be to run the Execve syscall with arbitrary arguments. This will involve finding suitable gadgets to create the call stack which would be used if the syscall was being made legitimately. We used a similar approach to both the one used by ROPgadget and in [4] which is based on using POP and MOV gadgets to move the required arguments to a writable section of memory piece by piece - more specifically gadgets of the form POP {SRC}, POP {DST} and MOV DWORD PTR [{DST}], SRC. With these, we can move (almost) any word-sized piece of data of of the stack and into memory by placing the respective data chunks after the POP gadgets and using the MOV gadget once the registers are populated. This can then be repeated for all requied arguments. Where the argument is a string, in this case the program to be called via Execve (e.g. /bin/sh), a NULL byte must be placed in subsequently to said string in memory. This can be achieved with the gadget of the form XOR {SRC}, {SRC} followed by the MOV gadget. In the case of the additional arguments for the program to be executed, we can do something similar - adjacent strings in memory separated by NULL words. However, this time

we also need a further stack to hold pointers to each of these arguments. We offset this stack in memory by 60 and build it using the aforementioned gadgets to move the address of each argument into memory, terminated with a NULL byte courtesy of the XOR method. This does introduce the limitation of only having arguments of less than 60 bytes (or whatever the offset is set to), but this should be sufficient for most programs. Once the call stack is constructed, we need to populate the required registers before using the interrupt mechanism to invoke execve. In this case, EAX will hold the syscall number (0xb); EBX the address of the first arguments; ECX the address to the array of pointers to each argument and EDX the environment parameters. We can use a XOR EAX, EAX gadget or the XOR {SRC}, {SRC} used in building the stack in the case {SRC} = EAX, to set EAX zero followed by iterative INC EAX gadgets to set it to 0x0B, to follow with a INT 0x80 gadget to invoke the interrupt.

3.5 Heaps of Shellcode

Extending on the Execve ROPchain, we designed functionality for copying shellcode into memory, which could then be turned executable via MPROTECT. We decided not to implement due to the requirement for either the writabl .data address to be page aligned or for privileged access to the binaries unused memory during run-time.

3.6 From Shellcode to ROPchain

As outlined in [2], malware consisting purely of ROP-gadgets poses a challenge to Antivirus software as it is effectively posing as legitimate code. In our tool, we built in the capability to compile a given Shellcode into its ROPgadget equivalent. It only accurately compiles a limited instruction set, and therefore very limited shellcodes but serves as proof of concept which could be extended.

We start by disassembling the Shellcode using the capstone library. In [2], more complex instructions are unrolled to their atomic equivalents which improves on our approach of using the instructions at face-value and therefore gives us less instruction-gadget pairings. Once we have our assembly instructions, we make a forward and then backward pass through the program to infer register access (registers to be preserved for after and before each instruction respectively), as well as compute the required stack size - we follow on from the approach in **3.2** by having a 'shadow stack' in memory. Next, we do another forward pass through the program evaluating each instruction in turn.

Many instructions can be simply mapped to a gadget by a string literal, however due to our shadow stack approach, instructions like PUSH need to be treated differently as we can't simply push a value to the stack is it will override our ROPchain. Instead we can use the POP and MOV DWORD gadgets as in **3.2** to build our stack in memory. Extending from this, any MOV operations involving the stack pointer i.e. ones which set other registers to point to different parts of the stack, can be replaced by popping the stack address in memory into those registers. For Example...

In the case of popping/pushing a small constant (i.e. a single byte), we can use a combination of POP and INC gadgets. Just popping the constant on its own could be thwarted by the presence of NULL bytes (strcpy will stop reading). Our gadget finder also failed to yield a self-XOR for every register. As such, we add a POP gadget, with the value 0xFFFFFFFF (-1) appended. Either the relevant INC or DEC gadgets can then be used to get the register to the any reasonably small value (positive or negative). We initially took the approach of XOR gadgets to clear registers, but these gadgets were often not available for all registers, whilst POP, INC and DEC often are.

We then have a table of instruction literals, with a snapshot of the registers which need to be preserved before/after the instruction in the scope of the entire program. We can then pass these all to our gadget finder as in **3.1** which will only return gadgets which maintain the correctness of the registers throughout the program.

4 Evaluation

Our tool was able to build ROPchains to successfully execute EXECVE with arbitrary commands and emulate two different shellcodes. This worked with multiple binaries with different buffer lengths and .data addresses.

4.1 Restricted Bytes and Stack Pointer Manipulation

Due to NULL terminating input functions such as STRCPY, we cannot pass any data or gadget addresses containing NULL bytes in our ROPchain. To address this, we aimed to find a general method of using arithmetic gadgets to manipulate the value of any register arbitrarily, through either ADD or SUB gadgets, various bitwise operators or a double-and-add approach (ADD REG, REG and INC REG gadgets). However, even in basic cases, we would often need to be able to either manipulate any register or manipulate a single register and then move this value into any other register, as an immediate and not a pointer to it - and our program could not consistently find the required gadgets to do both of these. In [5], Nurmukhametov et al. use a dynamic programming approach to solve this problem, keeping a stack of states and backtracking through the programs to find addends to use with the add-with-carry operation to produce

restricted values from unrestricted ones.

Given our program could often find gadgets to manipulate a single register to an arbitrary value, we developed a method of using PUSH gadgets to PUSH this value followed by pushing a pop gadget so that it is aligned before it. This would then be able to POP this arbitrary value into any register, with the caveat that this would need a gadget which could PUSH two registers in one sequence with minimal side-effects.

Although, it is possible to write NULL to a buffer using the `read()` function. `read()` does not terminate on NULL bytes, we believe that implementing a method to write NULL to the buffer using this function could expand the functionality of our program in the context of ROPchains, as well as opening avenues to bypass canaries in the stack, which almost always contain a NULL byte.

4.2 Gadget Selection

Our tool is efficient in its use of gadget selection, requiring only a few specific gadgets to build a shadow stack and make syscalls. However it loses adaptability by being heavily dependent on these predictable gadgets and a reliance on long chains of them renders it ineffective against countermeasures which look for exactly these. One could mitigate this by finding a way of recreating the key gadgets from longer more complex (**3.2**) ones which JMP to other gadgets - concise gadgets ending in RET make for more identifiable attacks [6]

There are a number of corner cases where our GadgetFinder could return gadgets which would produce erroneous behaviour. For example if the found POP gadgets for two registers side-pop each other, our Shellcode compiler may not function correctly as it would attempt to infinitely apply gadgets which would correct register.

A caveat of the gadget finding process in [2] is it omitting the replacement of operations which access the stack pointer with ROP equivalents. Ours differs by replacing these operations with equivalents for our shadow stack in memory - when compiling the ROPchain, we keep track of what our virtual stack pointer would be and use it as a replacement for ESP. For example the operation MOV EAX, ESP would be equivalated to popping the virtual stack pointer into EAX - the true value of ESP during execution is relevant only in that it moves through our ROPchain.

Whilst our method of searching for the least 'complex' gadgets makes our tool simpler more likely to compile a successful and correct ROPchain, using this complexity scoring system to find long convoluted gadgets could make our tool less likely to be stopped by countermeasures which search for less complex RET-ending gadgets.

4.3 Ability to convert Shellcode

Our solution can only compile shellcodes which consist of a small specific set of operations. It is, however very extendable - one only need add a case for each assembly instructions and ensure that suitable gadgets are found based on register preservation requirement. The reliance on on replacing single instructions (such as PUSH, POP) with multitude of gadgets (chains of POP, MOV and INC/DEC) means the converted Shellcode will almost always be somewhat larger than the original. In the case of particularly large shellcodes, we could find that our ROPchain causes a segfault simply by exceeding the allowed memory for the target binaries execution stack. The table below illustrates this size difference, as well as the fact that shellcode size has little bearing on exploit size; showing the large transformation involved in ROPchain compilation.

Shellcode	Size	Exploit Size
spawn bash	23 bytes	228 bytes
killall	11 bytes	272 bytes

4.4 Canary in the coal mine

An alternative avenue for improvement could be the implementation of the functionality to bypass stack canaries [7]. Canaries are safeguarding values after the buffer on the stack, they determine whether the buffer has been overwritten by checking whether the canary has changed before the `ret` of a function, and half the process if so. So far in our project, we have used gcc to compile 32 bit programs, so will keep that as the scope of our argument. We have been so far using `-fno-stack-protector` on compilation to disable canaries, but if we use `-fstack-protector` command or as default canaries will be introduced. In gcc, when `/dev/urandom` is available a random 4 byte canary is implemented (the first byte is always NULL for random canaries); otherwise, a terminator canary is used as an alternative. Canaries, by default, are implemented only in functions with buffers exceeding 8 bytes or calls to `alloca()`.

There are many techniques to bypass canaries, each of which depends greatly on the type of canary implemented. The largest hurdle to bypass canaries is the ability to write NULL to the buffer, this can be accomplished through use of the `read()` function as mentioned above. For random canaries, brute forcing the canary value is feasible on local machine attacks because there are 2^{24} possibilities of canary value; this method would not be feasible on 64 bit architecture. To improve this brute force method, the canary can be guessed byte by byte as if a single byte is wrong the

program will halt; decreasing the average number of guesses but the maximum number doesn't change. Another way to bypass a canary is to trigger an exception in the program after the overflow but before the canary instruction check.

5 Conclusion

In our project, we were able to build a program that, given an assumed buffer overflow vulnerability, automatically finds the string length that is necessary to overwrite the saved RET address. We also were able to automatically generate the exploit which takes arbitrary command line for execve and on a successful exploit, executes the command. It works for any given .data address. On top of this, we generate a ROP based exploit for a given arbitrary shellcode. We have also managed to, outside the scope of the assignment, develop a method to beat ASLR on 32 bit architecture.

6 Appendix

Individual Contribution Percentage:
50%

I developed the parts of the tool which searches for and parses required gadgets and builds a ROPchain for any given EXECVE call. I also developed methods for moving restricted bytes into required registers as well as building a compiler to turn different shellcodes into working ROPchains. I also designed alternative methods of running shellcodes despite non-executable stacks

References

[1] Jiang X. Freeh V. W. Liang Z. (2011 March) Bletsch, T. Jump-oriented programming: a new class of code-reuse attack. *In Proceedings of the 6th ACM symposium on information, computer and communications security (pp. 30-40).*

[2] Poulios G. Karopoulos G. tantogian, C. and C. Xenakis. Transforming malicious code to rop gadgets for antivirus evasion. *IET Inf. Secur., 13: 570-578.*, 2019.

[3] A. et al. Follner. Pshape: Automatically combining gadgets for arbitrary method execution. *Barthe, G., Markatos, E., Samarati, P. (eds) Security and Trust Management. STM 2016. Lecture Notes in Computer Science(), vol 9871. Springer, Cham. https://doi.org/10.1007/978-3-319-46598-2*$_1$5.

[4] Comsm0049 systems and software security github, Sep 21, 2022.

[5] V. Logunova A. Nurmukhametov, A. Vishnyakov and S. Kurmangaleev. Majorca: Multi-architecture jop and rop chain assembler. *Ivannikov Ispras Open Conference (ISPRAS), Moscow, Russian Federation, 2021, pp. 37-46, doi: 10.1109/ISPRAS53967.2021.00011.*, 2021.

[6] Tao ; Shi Yunxiu ; Li Ang Huang, Zhijun ; Zheng. A dynamic detection method against rop and jop. *2012 International Conference on Systems and Informatics, ICSAI 2012, 10.1109/ICSAI.2012.6223219*, 2012/05/01.

[7] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.