CS 576 Final Project

Gadget Finder for Exploit

December 11, 2020

Team 1
Ethan Grzeda
Matthew Jacobson
Joshua Mimer
Logan Rechler
Kaitlyn Sharo

Table of Contents

Note: We have changed nearly the entire design document for this third and final deliverable. To avoid highlighting the entire document, we wanted to note here that, given the updated requirements on Piazza and the grade on our original Design Document, we made this document completely from scratch and it bears minimal similarity to the previous submission.

Basics

Platform: Linux Lab - smurf host

OS: Linux Ubuntu OS

Architecture: x86_32 bit

Tools for Development:

- Dissassembler
    - Capstone Engine
- Changing memory permission
    - mprotect
- Debugging
    - GDB
- Programming language : Python 2.7

Files:

- gadget_finder.py
    - gadget_finder finds the necessary gadgets, compiles them into a proper ordering, and outputs a raw ROP payload file
- payload_tester.py
    - payload_tester combines the payload with the shellcode and runs that on the binary file.

High-Level Functionality

Input of gadget_finder.py:
The ROP payload generator gadget_finder.py takes as input at least one 32-bit executable ELF binary, the stack base address for that executable, and the binary base address for that executable as command line arguments.

Output of gadget_finder.py:
A corresponding number of payload files named payload_{binary} that are the raw ROP payloads for their respective binary files.

Input of payload_tester.py:
The payload tester payload_tester.py takes as input three command line arguments. The first argument should be the name of the binary file to exploit and the second argument

should be the name of the payload file and the third argument is the name of the file that contains the shellcode.

Output of payload_tester.py:
This outputs whatever your shellcode outputs.

Design Details

Gadget Collection

Note: We were only able to disassemble the binary at the level of the capstone disassembler because we were not aware of the requirement to find unintended gadgets with enough time to implement it.
To collect gadgets, we imported capstone into python. Capstone is a lightweight multi-platform, multi-architecture disassembly framework that has a simple API which allows for the disassembly of an X86 binary to print out the targeted assembly code.
The algorithm we used for gadget collection follows:

1. Open one of the binaries (must be in ELF format).
2. Use ELF tools to get each section of the code, then get the data from that section (the data is equivalent to the code for that section)
3. Then we disassemble the code using the capstone disassembler (md.disasm())
4. From that disassembly, we create three arrays of mnemonics (the command like "pop", "move", "ret", etc), op_strs (the registers being operated on like "edi", "eax"), and addresses (where that command is precisely found. To do so, we find the offset that the section is from the base of the binary using sh_offset. Then we add this to the base address received from the user.)
5. After we create these arrays, we search through each line of code that we gathered information from and look for the mnemonic "ret". Once we find "ret", we backtrack up to 5 steps (or until we find another "ret", whichever comes first) and save that set as a possible gadget.
6. Once we have saved all of the possible gadgets (sets of instructions that end with a "ret" command), we begin the search for the gadgets we require (see section "Payload Generation")
7. To do this, we loop through each possible gadget. We have a list that can hold each of the gadgets we need. We're looking for "pop eax", "pop ebx", "pop ecx", "pop edx", and "int 0x80".
8. We search through the mnemonics and op_strs of each gadget and find all possible gadgets that have a required command in it. We save the gadget, starting at this command, to the list we created. We have separate lists inside the larger list for each possible gadget. So we may find many possible gadgets that would work for "pop eax". All of these possible gadgets will be stored in a sublist in our gadgets list. Each of these sublists is then sorted by size (smallest first).

9. After we have found all of the possible gadgets for the ones we require, we need to find the best possible gadget. We start by checking if any of the gadgets are of length 2. If so, it means that gadget only contains the required command and the "ret" command.
10. Otherwise, we check if it contains a "mov", any form of “jmp”, or "lea". If it does, we call that gadget invalid because it can cause collisions, meaning that it can overwrite necessary registers (or change our location in the stack or some other error)
11. Next, we check for "pop" commands. There are two different types of "pop" commands: those that will affect the registers we need to fill (eax, ebx, ecx, and edx), and those that don't.
12. Every time we encounter a pop, we determine whether it is an address we need or one we don't. If it is one we need, we fill it with its required value, otherwise we fill it with 'A's. This way we maintain the integrity of our required registers. (see section "Maintaining Register Integrity" for more details on this process).
13. At the end, we have now found the best gadgets. They are the shortest, valid gadget available for each command we require.

Maintaining Register Integrity

Inside the function cleanGadgets, we check for certain commands in order to make sure the gadgets we use are valid. An invalid gadget may include a command such as "jmp" or "mov". There was also one specific command we were looking for as we searched these gadgets: "pop". A "pop" command can be detrimental to an attack that uses a ROPchain because it could pop an address from the stack that we intended to put in a specific register, or it could pop an address from the stack into a register we already filled with the correct value.

To mitigate this problem, we came up with an algorithm to filter through pop commands to figure out what to add to the stack so that each register gets popped information that fits it. We do not check the first or last commands during this algorithm because at this point we already know what they are.

This algorithm centers around an array formatted in the following way:

[gadget, [indexes with pop commands for other regs], [indexes with pop eax], [indexes with pop ebx], [indexes with pop ecx], [indexes with pop edx]]

We also need to define what we mean by "index" when we describe this algorithm. We find gadgets with a specific amount of commands. Each gadget will have no more than 6 commands. It may look like:

pop eax
pop esi
pop edx
ret

In this case, pop eax is at index 0, pop esi is at index 1, and so on.

For each "pop" command we find, we first check if the register called is one of the registers we care about (eax, ebx, ecx, edx). These four registers need specific values popped into them. If we encounter a command that pops into one of these four registers, we place the index of that

command into the corresponding array. Using the example above, we would get the following array

places = [gadget, [1], [], [], [], [2]]

From there, we feed this array into another function (ordArr). Since our main goal is to create a python file to help us create our payload, everything we do in ordArr centers around building a string. We loop through the "places" array and, for each index, we check where that index falls in "places", if it does at all. If the index falls in the first subarray (places[1]), we add 4 'A's to our string. This means that, when the program encounters this command, 4 A's will be popped into whatever register is noted there. If the index falls in one of the other subarrays, we add the correct information that should fill the specific register that corresponds with that subarray. This information comes from the "values" global dictionary which maps our required registers to their correct values.

At the end, this algorithm will ensure that if we are popping into a register in which we need a specific value, we always pop that specific value in and, if we are popping into another register, we fill it with bytes so that nothing gets popped incorrectly.

Payload Generation

Part of our project our task includes producing a payload that we will be using to perform our exploit. We generate a payload which is a file filled with the bytes that we gather throughout this process. It contains all the gadgets ordered properly to execute the ROPchain and return. To do so, we look for 5 specific gadgets. They are "pop eax", "pop ebx", "pop ecx", "pop edx", and "int 0x80". We take advantage of “int 0x80” for a kernel call to mprotect and fill the registers with appropriate values accordingly.

When calling mprotect, the first parameter in the ebx registers holds the page aligned address which the attacker will provide in our [insert name here] python script. After obtaining the address we fill the ecx register with the required length of our page. We want to change the permissions of the memory in which the shellcode already resides so we want to XOR PROT_READ, PROT_WRITE, AND PROT_EXEC which comes out to a hexadecimal value of 0x7. Finally to actually execute the call to mprotect, we must pass in the kernel call value associated with that function. In the case of mprotect, the kernel call is 0x7d and this value will be popped into the eax register.

While we are not flexible with what gadgets we are looking for (we must find all 5 of these specific gadgets), we are flexible about what they contain. The gadgets can be up to 6 instructions long and can contain anything other than any sort of mov, lea, or jmp instructions. We chain these instructions by first using the “pop ebx” gadget, then the “pop ecx” gadget, then the “pop edx” gadget, then the “pop eax” gadget, and finally the “int 0x80” gadget. Finally, we input all of these instructions into a dummy python file which then prints them into a payload file.

Payload Tester Design

The payload tester is a simple python script that uses the name of the binary, name of the payload file, and the name of the file that contains the shellcode to run gdb on the binary with the payload is the argument. The payload tester opens up the payload file and reads it in. Then it reads in the shellcode and combines that with the payload. Finally, our tool uses the OS API to run command line commands using psopen() calling our script while piping our output into our gdb call and printing the result to the console for easy viewing.

Code Outline

Global Variables:

- The three following variables line up to create the idea of an address that matches with a mnemonic and op_str. For example, "pop eax" is at address 0x54321
    - addresses - a list containing all of the addresses of every valid line of code in the binary
    - mnemonics - a list containing all of the commands (pop, push, mov, etc.) in the binary
    - op_strs - a list containing the op_strs (ebx, ecx, etc.) for all the commands in the binary
- notSafe - the list of registers that cannot be modified due to our implementation of the ROPchain
- values - a dictionary that maps the required values to the registers

Functions:

- openBin
    - Input: a binary file name
    - Output: the file descriptor of the binary
- readBin
    - Input: a binary file in ELF format and a section name to read
    - Output: the code from the specified section of the file
- sortCode
    - Input: a file name
    - Functionality: Initializes the global variables (addresses, mnemonics, and op_strs) by reading through each section of the file and disassembling it.
- findGadgets
    - Input: the maximum length for the gadget to be
    - Output: a list of all possible gadgets with size <= maxLen
    - Functionality: For each line of the assembly code, see if the command is a return. If it is, start adding the command as a gadget. Backtrack maxLen positions or until another return is hit. Then, it adds each set of these commands as a gadget.
- findCommand
    - Input: a full command to look through); a string mnemonic to look for (i.e. "pop"); a string op_str to look for (i.e. "eax")

    - Output: True if the command has the given mnemonic and op_str, False otherwise
    - Functionality: This is helpful when we look for our required gadgets. It allows us to easily check each command to see if it matches what we want.
- sortGadgets
    - Input: a list of gadgets
    - Output: a list of lists containing gadgets that are possible for each command
    - Functionality: This goes through the master list of gadgets we found and checks if any of them are the required assembly commands. If they are it adds them to a list of possible gadgets for that respective assembly command. Then it sorts that list so the shortest gadgets are always at the start of the list.
- cleanGadgets
    - Input: the list of lists of gadgets that sortGadgets output
    - Output: the final list of the shortest, valid gadgets we can use
    - Functionality: This gadget picks the "best" gadget for each required gadget from the list. To do so it creates a 2d list object to put the gadget in (see Maintaining Register Integrity for more information on this object). We first check to see if there's a gadget of length two. This would mean that it is the instruction we want(i.e. "pop eax") directly by a "ret" which is always the best option. If there are no gadgets of length 2 then it looks at the current smallest gadget and runs some checks on it. If there is a pop it checks to see what it is a pop of. If it is a pop of one of the gadgets we need then it adds the position of that instruction within the gadget to its spot within the object. If there is any kind of mov, lea, or jump instruction we consider the gadget bad because it can change the registers or the instruction pointer. Therefore we throw that gadget out and move to the next smallest one. When we find a valid gadget we stop searching through the list of possibilities for that gadget and move onto the next list of possible gadgets. Finally, we return the list of the shortest valid gadgets that can be used.
- ordArr
    - Input: the 2d gadget object and which gadget it is
    - Output: a string to add to the intermediate python file that will be used to create the payload
    - Functionality: This function creates the output from the gadget object and appropriately adds values after it to be popped. (see "Maintaining Register Integrity" section for more details)
- generateAttack
    - Input: the final list of gadgets (from cleanGadgets) and the name of the binary
    - Output: prints the intermediate python file that is used to create the payload to the screen
    - Functionality: creates a python file that forms an exploit. Turns the python file into a payload using "python {file} > {payload_name}" through the os.popen command. Prints the python file to the screen, saves the payload, and deletes the intermediate file.

Main Function

- Functionality: Our main function first does error checking such as ensuring that the user has inputted the command line arguments correctly and that the files inputted exist. Next, it attempts to find gadgets in all of the binaries inputted and outputs an error if gadgets cannot be found in one or more binaries. After finding the gadgets, generateAttack is called to create a payload for each binary for which gadgets were found.

Testing

We were able to test our payload on a 32-bit binary that we got from a previous lab assignment (so-32). This binary met all our requirements for our payload because it did not include any unintended read calls such as strcpy() which was not working for us as planned due to the fact that strcpy() does not process null bytes. Null bytes were found in our shellcode and registers from the pop commands. In order to test on the so-32, we saw that a buffer overflow attack was required to exploit this specific binary. After padding the correct amount of A's and putting the appropriate hex values in each register, we then located the address at which the shellcode was located for the desired execution. We also took advantage of the info proc mappings command in gdb in order to simulate a user inputting the binary and stack base address of the binary in order to create the effective ROPchain.

Constraints

- This exploit generator may not work if there are not enough gadgets in valid code.
- The outputted exploit may not work with strcpy() as it will likely contain null bytes. Strcpy() does not read null bytes and therefore the exploit may become misaligned when executed.
- We were unable to account for every possible instruction that may make a gadget invalid (i.e. pushing a value to the stack and popping it to our required registers), so occasionally an exploit may contain a gadget that causes the program to break.

Instructions

1. First, ensure that all of the necessary dependencies are downloaded. From the Linux terminal, type the following lines:
    a. Python - sudo apt install python2.7
    b. gdb - sudo apt-get install gdb
    c. ElfFile import - pip install capstone pyelftools
    d. ElfTools - pip install pyelftools
    e. Capstone - pip install capstone
2. Once everything is downloaded, place gadget_finder.py, payload_tester.py, and the binaries you would like to exploit in the same EMPTY directory.

    a. Please note that this directory should be empty to avoid files being overwritten or deleted during execution
3. Navigate to that directory in your Linux terminal by using the `cd` command.
    a. Ex. if the directory you chose is /usr/exploit/secret, type in the command line `cd /usr/exploit/secret`
4. TO RUN gadget_finder.py: (basic usage: `python gadget_finder <binary> <page aligned address> <binary base address>...`)
    a. In the terminal, start by typing `python gadget_finder`
    b. Follow this by the name of the binary you want to exploit, the page aligned address, and the binary base address. For example, if the binary is named to_exploit.bin, the entire line should read `python gadget_finder to_exploit.bin 0xfffde000 0x0804e000`
    c. If you want to exploit multiple binaries, simply put the name of each binary with the appropriate addresses after `python gadget_finder`, followed by a space (`python gadget_finder to_exploit.bin 0xfffde000 0x0804e000 vuln_prog.bin 0xfff7e000 0x0808f000...`)
    d. Hit the enter key to run the program
        i. Troubleshooting: if this throws an error from the terminal, try using `python ./gadget_finder` instead. Still doesn't work? Try `python2 gadget_finder`. If it still isn't working, make sure you have all of the dependencies from step (1) downloaded
5. Understanding the output of gadget_finder.py:
    a. For each binary you input to gadget_finder.py, you will see an output in the terminal. If the output is an error:
        i. FileNotFoundError - the binary could not be found
        ii. TypeError - the binary could not be read
        iii. Error (code1) - there were not enough gadgets in the binary to work with.
        iv. Error (code2) - the program found the basic instructions to work with, but a command appeared within that gadget that rendered it invalid.
    b. Otherwise, gadget_finder.py will add a payload file to your current directory for each binary.
6. TO RUN payload_tester.py: (basic usage: `python payload_tester <binary> <payload> <shellcode with location>`)
    a. This can only run with one binary at a time. As input, give it the name of the binary, the name of the payload (created by gadget_finder), and a file containing your shellcode and the location of that shellcode (formatted <location of shellcode><shellcode>). This will execute your exploit.

Example:
Assuming binary vuln_prog.bin in directory /usr/shared/exploit
`cd /usr/shared/exploit`
`python gadget_finder vuln_prog.bin 0xfffde000 0x0804e000`
(At this point, if no errors were thrown, `ls` *will show you payload_vuln_prog in the cwd)*
`python payload_tester vuln_prog.bin payload_vuln_prog shellcode`

Link to Demo Video

https://kaltura.stevens.edu/media/1_10j80ne9

I pledge my honor that I have abided by the Stevens Honor System.
Ethan Grzeda, Kaitlyn Sharo, Matthew Jacobson, Logan Rechler, Joshua Mimer