CSE 543 - Fall 2018 - Assignment 3: Code-Reuse Attacks

1 Dates

- **Out:** *November 11, 2018*
- **Due:** *December 6, 2018*

2 Introduction

In this assignment, you will produce a few code-reuse attacks. The attacks include a traditional buffer overflow, a basic heap overflow, and a disclosure attack exploiting a weakness in `strncpy`.

3 Background

Code-Reuse Attacks Code-reuse attacks exploit the available code to perform exploits. Such attacks leverage a vulnerability that enables an adversary to change control data, such as function pointers and return addresses, to direct program execution to code chosen by the adversary. As described in the Return-Oriented Programming paper of Roemer (see 9/20/18), an adverary may still create malicious behavior by reusing the victim's code. Examples include running the "system" library call provided by `libc`.

In this project, we will modify return addresses and function pointers to direct code to execute these functions by calling their implementation in the *Procedure Linkage Table* (PLT).

Procedure Linkage Table (PLT) The PLT provides some useful options for launching code-reuse attacks. The PLT defines the stub code for invoking the library calls used by the executable. Since library code does useful things, such as invoking system calls, invoking this code via the PLT is often useful to the adversary. You can view the PLT stub code by disassembling the executable.

```
objdump -dl cse543-p2 $|$ less
```

You can then search for "plt", such as `system@plt` and `strncpy@plt`, to locate the stub code for a variety of library calls from the executable code. We will mainly use that code in our exploits.

Buffer Overflows The idea in all the code-reuse attacks in this project is to overflow buffers to enable the modification of memory used in control flow decisions. In part 1, we will overflow a return address. In part 2, we will overflow a function pointer in a structure. In part 3, we will overflow a pointer to a structure to set it reference a function pointer, enabling us to modify the function pointer.

In this project, you will be given the source code and a binary for an extended version of the server program from the previous two projects. This server implements two new operations `CMD_WRITE` (in function `receive_write`) and `CMD_READ` (in function `receive_read`). `CMD_WRITE` operation enables the creation of a structure in the server, and `CMD_READ` enables reading of a field in that structure. You will exploit vulnerabilities in `receive_write` to implement your attacks.

4 Example - Stack Overflow

To learn how to launch such exploits, you should construct an exploit payload to overflow the return address on the stack for the `receive_write` function to redirect the program to run the `/bin/ls` command. This task is not to be handed in, but to teach you some of the key steps in exploit generation.

In this exploit, you craft an input file to be sent from the client *cse543-x1* to the vulnerable server *cse543-x1-server*, which processes the input file in the function *receive_write*. You will overflow the buffer `buf` at the `memcpy` command in the function (there is only one). To submit the exploit payload to the victim, run the server as usual

```
./cse543-x1-server <private-key> <public-key>
```

To perform the exploit, you need to: (1) find the distance from the buffer start to the return address; (2) find the addresses for the code and data you want to reuse; and (3) construct the exploit payload using this information.

Find distance to return address. Then, invoke the client with the exploit file (more on building that below) to run the `CMD_WRITE` option 3, in the third argument.

```
./cse543-x1 <exploit-file> <server-ip> 3 1
```

To perform a buffer overflow, you must determine the distance from the buffer your want to overflow (`buf`) and the return address. To do this run the program under *gdb* and set a breakpoint for the `receive_write` function. To find the address of `buf` on the stack (it is a local variable), print its address in *gdb* via

```
p &buf
```

See `http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf` for more information on gdb commands. To find the return address, find the address of the instruction after the call instruction to `receive_write` in the binary using `objdump` as above. Once you have the address (write it down), you need to locate the address on the stack. To display the stack starting at `buf`, perform the following *gdb* command.

```
x/200wx <buf-addr>
```

Since `buf` is a large buffer, the return address must be beyond the end of the buffer. *Record the distance (number of bytes) between* `buf` *and the return address.* As the binary is compiled in 32-bit mode, each address is four bytes.

Find addresses of code to reuse. To execute `/bin/ls`, you need to find the addresses for the `system@plt`, `exit@plt`, and the string "/bin/ls." The first two are the PLT stubs for the `system` and `exit` library calls, respectively. The `system` library call executes the file specified as its argument in a shell (/bin/ls in our case). These PLT stub addresses can be found in the `objdump` as constructed above.

The string "/bin/ls" is provided in the binary. You can find the address of this string using the `strings` command.

```
strings -t x cse543-x1-server
```

You will have to compute the actual address of the "/bin/ls" string in the binary by adding the offset from the base address of the executable. This can be found in `/proc/<pid>/maps` while `cse543-x1-server` is running.

Construct the exploit payload. In the project tarball, a file is provided as a shell of a program to construct an exploit payload. The exploit payload must: (1) overflow the buffer up to the return address; (2) replace the return address with the address of the function you want to call (`system@plt` PLT stub); (3) place the address of `exit@plt` next to exit cleanly; and (4) place the address of the start of the `/bin/ls` string after the exit entry.

Write this exploit payload in the file `cse543-stack-attack.c`. This file uses `memset` to create an input of size `RTN_OFFSET` (number of bytes from start of `buf` in `receive_write` to return address). The function `write_to_file` loads data into a file (see `cse543-util.c`). The macro `pack` transforms addresses into little endian form to stuff binary addresses into the payload easily.

You are given a shell of this file, where you will need to set the `RTN_OFFSET` value, the addresses in the `pack` commands, and the amount of data to submit in the `write_to_file` operations.

Build `cse543-stack-attack.c` via `make stack-attack` to create the binary `./stack-attack` that creates the file `stack-payload`. Check that the value of `stack-payload` is what you expect before running it, as below.

```
./cse543-x1 stack-payload <server-ip> 3 1
```

5 Project Tasks

In the project, you will perform two exploits. You will receive personal binaries and source code for performing each of these exploits.

5.1 Heap Overflow Exploit

In this task, you construct an exploit payload to overflow a function pointer in a structure allocated on the heap in the `receive_write` function to redirect the program to run the `/bin/sh` command.

In this task, you will overwrite the function pointer in the structure provided to you in `cse543-format-X.h`, where `X` is a small integer indicating your individual structure. Each `struct A` definition includes a field for `op0`, which is a function pointer. The aim is to overwrite `op0` with a pointer to the `system` PLT entry and invoke `op0` in a manner that executes `/bin/sh`.

Overwrite `op0` field. In the executable, `op0` is initially set to `strcpy`, and it is used to perform each of the string field updates for `struct A` fields. Since `strcpy` has no bounds checks, then overwriting the end of a string field (each is limited to 16 bytes) is straightforward. Use the gdb methods above to determine the distance between fields in the structure for the structure object `a` to determine how to pad strings to overwrite `op0`.

Run the new `op0` function. Once you overwrite `op0`, then its next use will invoke `system` (or whatever you set it to), so you should be ready to invoke `/bin/sh` by preparing the arguments according. This will involve modifying other fields in `struct A`. Note that you are simply invoking `op0` via the call instruction, so you can invoke by setting the arguments for the call statement you want to use in the code.

Construct the exploit payload. In the project tarball, a file is provided as a shell of a program to construct an exploit payload. The exploit payload must (not necessarily in this order): (1) use an overflow of the `op0` field to replace the `op0` field with the address of the function you want to call (`system@plt` PLT stub) and (2) enable execution of the `/bin/sh` command upon an invocation of the `op0` function pointer.

Use the file `cse543-heap-attack.c` to construct the exploit. This version handed out shows how to build a payload to perform normal string and integer field updates for `struct A` as implemented by

cse543-format-X.c. The function write_to_file loads data into a payload file (see cse543-util.c). To write a memory address into the payload, you can use the macro pack as for the buffer overflow example. You will need to determine the string values to write to overflow the buffer up to the op0 field, write the desired address into the op0 field, and invoke op0 to run your exploit.

Note that full credit will be given if you have no more than one write to each field in struct A**.**

Build cse543-heap-attack.c via make heap-attack to create the binary ./heap-attack that creates the file heap-payload. Check that the value of heap-payload is what you expect before running it, as below.

```
./cse543-x1 heap-payload <server-ip> 3 1
```

5.2 Disclosure Attack

In this task, you construct an exploit payload to leverage a disclosure attacked enabled by strncpy to launch an attack to execute /bin/sh. This version of the binary differs in two key ways from the *Heap Attack* binary: (1) the binary uses strncpy to update structure fields in struct A in the receive_write function and (2) strncpy is stored in a function pointer array (funcs) on the heap. Using strncpy prevents the easy buffer overflows allowed by using strcpy, but is still prone to buffer overread attacks because it fails to guarantee null termination.

In this attack, you will produce two exploit payloads, one ahead of time and one at runtime by building a payload file from the program execution using the disclosed information. The first payload will setup the victim binary for the buffer overread to disclose the address of the structure a on the heap, and the second payload will exploit the knowledge learned from the disclosure to modify a function pointer in the function pointer array funcs also stored on the heap to execute /bin/sh. Note that Linux uses ASLR to move the heap base pointer, but the offset between the structure a and the function pointers funcs remains constant, so once we know the address of a, we can exploit a buffer overflow to modify funcs. Bwahaha (evil laugh sound)!

Disclose structure a **address.** The structure struct A contains a field base that is set to the address of the structure instance a. NOTE: While this is contrived, structures often contain pointers to information on the heap that, if disclosed, enable attacks on heap data. The first challenge is to create a buffer overread that enables an adversary to obtain the value of this base field and hence an instance of structure a. T

To do this, one must craft a string for writing to structure a that does not include a null terminator. In addition, all integers between that string and the base field must not have a null byte — otherwise, the buffer overread will be terminated.

Use the file cse543-strn-attack.c to construct the file to prepare an instance of struct A for buffer overread. This file should be similar to the file cse543-heap-attack.c, except you will not be allowed to overflow the 16 byte buffers (unless there is another bug). I want to see you prepare the structure, so that a subsequent CMD_READ operation on a particular field string_f will also return the value in the base field. Build cse543-strn-attack.c via make strn-attack to create the binary ./strn-attack that creates the file strn-payload. Check that the value of strn-payload is what you expect before running it, as below.

```
./cse543-x1 strn-payload <server-ip> 3 1
```

Perform the buffer overread. The buffer overread will retrieve the value of the field string_f and other data including the value in field base of the structure you just added. You will need to implement the function build_malread to submit the CMD_READ request and receive the response.

Note that the server will encrypt the response from the CMD_READ request.

To invoke CMD_READ submit the following request. The "file" name identifies the payload file to generate as a result of the read request.

```
./cse543-x1 strn2-payload <server-ip> 2 1
```

Construct the exploit from overread data. Use the data retrieved from the overread address in base to compute the address of funcs in the payload. Since the offset between funcs and a are consistent (if they are not you are probably computing something incorrectly), then you can compute the offset even though the base address of the heap changes on each run. To help, I print the addresses of a and funcs and the offset from on the server, so you can double on the client. If you get different addresses, then you have made a mistake.

You then need to construct an offset using the pack and write_to_file functions/macros previously to create the file strn2-payload. In this exploit, you need to (not necessarily in this order): (1) leverage a stack overflow from buf in receive_write to overwrite the address in the variable a to point it at funcs instead; (2) perform a *type confusion* attack to modify array elements in funcs to change funcs[0], which is set to strncpy to system; (3) perform another *type confusion* attack to store the string "/bin/sh" in "a"; and (4) use code in receive_write to execute system (instead of strncpy) from funcs[0] on a field in "a" to execute /bin/sh.

Launch the exploit payload. Once you have constructed the exploit payload in strn2-payload correctly, you can launch it under a CMD_WRITE operation to update a in the malicious ways described above.

```
./cse543-x1 strn2-payload <server-ip> 3 1
```

6 Questions

1. How would you prevent the buffer overread necessary to implement the disclosure attack? Be specific about the fix.
2. How would you prevent the buffer overflow that exploits the disclosure attack? Be specific about the fix.
3. What is the best way to ensure safe string processing in the C language?

7 Deliverables

Please submit a tar ball containing the following:

1. Your exploit programs - cse543-heap-attack.c, cse543-strn-attack.c
2. Your dynamically generated payload - strn2-payload
3. Your modified cse543-proto-p3.c file, with build_malread function
4. Answers to project questions

8 Grading

The assignment is worth 75 points total broken down as follows.

1. Answers to three questions (15 pts)
2. Packaging of your attack programs using “make tar” and inclusion of that tar file and the questions in the tar file you submit. Your attack programs build without incident. (10 pts)
3. Heap attack (15 pts)
4. Disclosure attack (35 pts)