Casper CTF Report: Solutions

Joe Mulvany

December 15, 2023

1 Casper3 solution

1.1 Description

Casper3 is a program which reads an input from the user, which is a pin number, and checks if it is equal to the randomly generated pin. There are 4 important variables in this program, all allocated on the stack:

- A buffer pin of 4 characters, intended for the generated pin,
- a buffer input of 5 characters (with one byte reserved for the null byte) intended to contain the input of the user,
- an integer i used for the loops in the program,
- and an integer correct indicating if the pin is correct. The program first generates a pin based on the current time. Next, the user is asked to input a pin number.

1.2 Vulnerability

This program contains a vulnerability because the scanf call on line 20 does not limit the length of the input the user provides. This allows the user to provide a longer input than fits the size of the buffer, and thus overwrite other data in the stack.

1.3 Exploit description

To exploit this vulnerability, we provide an input longer than the input buffer. More specifically, we will provide an input that will overwrite the pin in the pin buffer.

The input we will provide the program is 111101111. This will write 11110 in the input buffer and 1111 in the pin buffer. This will make the first four characters of the two buffers equal, which is the condition we need to achieve to launch the shell.

1.4 Mitigation

This vulnerability can be prevented by replacing the vulnerable scanf call: scanf("buffer will never overflow. Other possible defenses are . . .

2 Casper5 solution

2.1 Description

The Casper5 program reads a user input from argv[1], entered when the program is called. This input is used to create an instance of the User struct, with the name variable set to the users input and the role variable set to the "defaultRole" which has authority 0. The program then greets the user and checks if their authority is equal to 1, if so the shell is launched. There are 5 important variables in the program:

- The User_ struct variable with members name and role_t,
- name is a string of length 775,
- role_t is a ponter to a role struct, which has members rolename and authority,
- authority is an integer,
- rolename is a string of length 32
- defaultRole is a global instance of the struct variable role_t with rolename set to regular user and authority set to 0

2.2 Vulnerability

This program's vulnerability comes in the form of a strcpy() function call in the inputUser definition. The strcpy function is unsafe as it does not perform bounds checking while copying a string from source to destination, allowing users to give inputs larger then the size of the name variable and thus overwrite other data on the stack.

2.3 Exploit description

To exploit this vulnerability the program is given an input larger than the size of the name variable, overflowing the string to reach the address of the role variable. The role variable currently points to the address of the defaultRole global variable, however the defaultRole variable itself is not a concern as the main goal of the exploit is not to change the role_t struct itself but to instead alter the address that role points to, so that the authority member of the role_t struct can be altered.

To do this we take note that due to the length of rolename, the authority variable's address is offset by 32 bytes from the base address of the role_t struct variable. If an address can be found elsewhere in memory that contains an integer one, i.e 7 zeroes followed by a 1 in hex, and the 1 is static in this address, i.e it's position is not dependent on what program or program instance is running, we can overwrite the role_t pointer to point to 32 bytes before the adress of this 1 integer. This will overwrite the authority variable and set it to 1 allowing us to pass the authority check and launch the shell. The one that I found was located in the c library and the final exploit was called using: /casper/casper5 'perl -e 'print "A" x 776 . "\xd4\xaa\xea\xb7"' to capture the flag: 4PxsjgRg3q7t81tSjUiUvKD78yPh7Jal.

2.4 Mitigation

This vulnerability can be prevented by replacing the vulnerable strcpy call with:

```
if (strcpy_s(name, sizeof(name), s) != 0) {
    printf("failed due to invalid user input");
}
```

Strcpy_s will ensure that the copy operation on the users input, s, does not exceed the size of name and if it fails due to the input being too large it will return a non-zero value, triggering the error message and cheaply and effectively preventing the user from overflowing the buffer/exploiting the program.

Another method of mitigating this attack would be to use Address Space layout randomisation (ASLR), which randomises the assignment of locations for various components within the processes memory address space every time a program is loaded into process memory. This method would change the base addresses of the stack, the heap, executable code, and libraries meaning that an attacker could no longer use the address of the integer 1 that they previously located and thus cannot suitably overwrite the authority variable.

3 Casper7 solution

3.1 Description

The casper5 program prompts a user to enter their name, reads the name into a string variable, buf, and prints a greeting back to the user with their chosen name. There is one important variable here, the buf string, which has a length of 775.

3.2 Vulnerability

This program contains a vulnerability because the gets() function call in the greetUser definition. Gets does not perform bounds checking while reading

characters from the standard input and will continue reading them into a buffer until a newline character or end of file is reached. This allows users to give inputs larger then the size of the buf variable and thus overwrite other data on the stack.

3.3 Exploit description

The key to this exploit is that the stack canary is disabled, allowing us to overwrite the return address and thus conduct a return to libc attack. The goal of all of our attacks is to run the script at /bin/xh, and so our first step was to set an environment variable equal to "/bin/xh" using the "export =" command, so that it could be called up later. Next the address of the standard library system() function, which takes a pointer to a string as an argument and then passes this string to the command interpreter to execute said command, was found using gdb and this address was recorded.

The address of the return call was then found by overflowing the buffer with larger and larger inputs until a segmentation fault occurred during the return call due to the corrupted address. With a method of overwriting the return address and the address of the system() function found, the remaining step was to locate the address of the environment variable, SHELL2 in this case, that we had set. The issue here is that the address of the environment variable changes between program runs and so a separate c program, getenv, was created to find the address of SHELL2 and return it. This function could be called within the same command that executes the rest of the exploit, preventing the address of SHELL2 from changing.//

The address of SHELL2 was offset inside of the getenv program to isolate the string "/bin/xh" and this address was combined with the 787 overflow characters, the address of the system function and the address of the exit() function to create a user input that will launch the desired shell. The address of the exit() libc function is used as a return address for the system() function which will prevent a seg fault from occuring and make the exploit harder to detect. The final command called, shown below, returns to flag: LWymYeyUJtPXUsmAOa5rtIG4ZhagJHip

export SHELL2="/bin/xh" && gcc -o getenv geten4.c && PIPE=$(./getenv SHELL2) && python -c "print 'A' * 787 + '\x50\x72\xe2\xb7\x20\xa4\xe1\xb7' +'$PIPE'" — /casper/casper7

3.4 Mitigation

This vulnerability can be prevented by replacing the vulnerable gets call with:

```
if (fgets(buf, sizeof(buf), stdin) == NULL) {
    printf("Error occurred");
```

Fgets() will read input from the user until a newline character is reached or if the size of buf is exceeded, cheaply and effectively preventing the user from overflowing the buffer/exploiting the program. ASLR could also be used to protect the program from being exploited as it would result in the base address of the c standard library being changed and thus the system() function's, or another potentially abusable function's, address moving, making it harder to access by the attacker.

The placement of a stack canary before the return address would mitigate exploits to the program, prohibiting users from overwriting the return address and thus preventing the program counter being hijacked to point to a lib-c function or gadget. Control Flow Integrity (CFI) could also be used to thwart a return to lib-c attack. CFI works by enforcing the legitimate control flow of a program by restricting the set of valid targets, function calls or jumps making it more difficult for an attacker to divert the execution flow to an unexpected location. In casper7 this would mean that that CFI would flag the diversion of the program to the system() function, throwing an exception or terminating the program in order to stop a potential attack.

4 Casper8 solution

4.1 Description

The Casper8 program reads a user input from argv[1], entered when the program is called. This input is passed to the greetUser function where it is copied into a buffer along with a greeting message, which is then printed back to the user, The program then checks if the variable isAdmin is set to 0 and if it is, return 0 will be called, if not the shell script will be run.

4.2 Vulnerability

The vulnerability in this program is caused by the use of printf without explicit format specifiers, meaning that the users input will be interpretted as the format string. The user can therefore supply format specifying characters such as (%s, %x etc) to the input and use this to read and write data to the stack.

4.3 Exploit description

The first step of the exploit is to get the address of the admin variable using gdb/grep. Next we find the memory offset of the printf argument, our input, by supplying the input "AAAA%x.%x.%x.". Since no argument variables were supplied, this input will cause printf() to search the stack for an integer value to print for every %x. format specifier included in the input, printing the hexidecimal representation of the integer value stored at each memory address it checks.By increasing the number of %x. format specifiers in the input, more

and more of the stack can be viewed, until the AAAA at the start of the input string, 0x414141 in hex, can be seen in memory.

In this program the position of AAAA was 46 bytes offset from the address that printf first read values from, an extra character was added before the AAAA input, e.g. BAAAA, so that the four A characters were now offset from the base by exactly eleven words. With this offset information a new input was created, consisting of one offset character, then the address of the isAdmin variable and finally %11$n, all together this input looks like: "B\x9c\x99\x04\x08%11$n". This final part of the input, %11$n, will write the number of characters printed so far, in this case B, to the address found at the 11th argument position and since we stored the address of isAdmin at this 11th argument position by including it in our input, the character B will be written to isAdmin, making it not zero and thus bypassing the admin check and return the flag: LVWaxTdmiJZRn4dWAEGlMqstk4aVAp3P.

4.4 Mitigation

Format string vulnerabilities can be prevented by using explicit format specifiers in the printf() function, i.e avoiding user controlled format strings. In Casper8 this would mean replacing the current greet user function with:

```
void greetUser(const char *s) {
    printf("Hello %s!", s);
}
```

By directly printing the user provided string using %s we ensure that the program interprets the user input as a string to be printed and not a format specifier, removing the potential for an exploit.

5 Web challenges

5.1 First SQL injection (exploit-sqli-0.sh)

5.1.1 Description

The first SQL injection exploit takes place at the login page of the Graiding website. Here there are two fields for the user to enter their email address and password, these two inputs will then be passed to an SQL Query of the form; "SELECT * FROM users WHERE email = 'email' and password = 'password'" to check if the user's details match those of a registered user in the Database.

5.1.2 Vulnerability

The vulnerability in this case arises from the lack of input validation and sanitisation. Users are allowed to input escape characters, with the commenting

characters "–", "/*" and "#" being most dangerous in this case, that could be interpreted as SQL commands, allowing them to bypass the password check and log in without being registered as a user, if they know the email of an existing user.

5.1.3 Exploit description

This exploit begins with finding an existing user email, one such email, west.ada@example.com, is linked to in the "Services" page of the Graiding site, underneath a customer review. With this user email, an input can be constructed that comments out the password checking element of the original SQL query, possible because not all escape characters have been blacklisted. In my exploit the input "west.ada@example.com'/*" was used. Here the escape character "/*" opens a multi line comment, thus making it so that the Query will only check that the email address matches a registered email before granting access to the rest of the site. The flag 911dc9f7d57cd4a9413b66e17b92e18ea3063e69 is returned.

5.2 Second SQL injection (exploit-sqli-1.sh)

5.2.1 Description

The second SQL injection exploit takes place at the post searching field on the course pages of the Graiding website. Here there is a field for the user to enter a string that will be checked for similarity against the titles and content of posts in the course page. This check is made using the SQL Query of the form; "select * from posts where (title like '%input%' or content like '%input%') and course_id = 2. This Query will return a post if one is found that is similar to the user's input and return "course not found" if not.

5.2.2 Vulnerability

Again the vulnerability in this case arises from the lack of input validation and sanitisation. Users are allowed to input escape characters that could be interpreted as SQL commands, allowing them to comment out parts of the SQL Query and insert their own conditional statements that will always be evaluated as true, bypassing the check that ensure users input are similar to an existing post and the check that the post belongs to the course that the user is currently viewing.

5.2.3 Exploit description

This exploit begins by inserting inputs that cause errors in the SQL Query, revealing what escape characters are blacklisted, in this case the error message below appeared: "Query cannot contain: LIKE, UNION, SELECT, ; or -". Next I added a quotation mark to close the input string and tried different inputs until an error that revealed the SQL Query appeared "AN ERROR OCCURRED: select * from posts where (title like '%input'%' or content like

'%input'%') and course_id = 2". With this information i was able to construct the following input "asasa%' OR 1=1)#". This works by first creating a statement that always evaluates as true and then commenting out all conditions after this by opening a multi line comment with the # escape character. This input will successfully return all posts from all course pages and gives the flag 12493c9542aaf9f7e4514f8ee664aa3db5fc9064.

5.3 Third SQL injection (exploit-sqli-2.sh)

5.3.1 Description

The third SQL injection exploit takes place on the course enrollment page of the Graiding website. Here there is a field for the user to enter the secret code their teacher gave them, allowing them to enroll in the corresponding course. The input is used to create an SQL Query of the form SELECT * FROM courses WHERE join_code = 'input', which will enroll the user in a course if one is found and return a course not found error message if not.

5.3.2 Vulnerability

Again the vulnerability in this case arises from the lack of input validation and sanitisation. Users are able to input escape characters that could be interpreted as SQL commands, allowing them to insert a UNION SELECT command that can be used to access parts of the database that are not intended to be accessed in the original query.

5.3.3 Exploit description

Again his exploit begins by inserting inputs that cause errors in the SQL Query, revealing what escape characters are blacklisted, in this case the error message below appeared: "Query cannot contain: OR, AND, LIKE, LIMIT, -, =, ;, true or false-". With this information I recognised that a Union select command that returned all results could be added with the input: ' UNION SELECT * FROM courses WHERE join_code <> 'abc . Here the Where join_code <> 'abc always evaluates to true, however this input generated another error message:

SOMETHING WENT WRONG: More than one record returned using SELECT * FROM courses WHERE join_code = '' UNION SELECT * FROM courses WHERE join_code <> 'abc'

With this constraint in mind I tested other conditions for the where command to narrow down the range of possible exploits, the condition WHERE join_code >'z' resulted in the error "course not found" and the condition WHERE join_code >'a again returned the more then one record error. This meant that some lowercase letter between a and z would result in just one record being returned and so using trial and error i identifed that the command ' UNION

SELECT *FROM courses WHERE join_code ¿'v, successfully enrolled me in a course and returned the flag: f7421d4e15150dc22bea9e96d388c1edde8095ca.