Ada Lerner

YYYY/MM/DD

**Exploit 0**

**(1-2 sentences) What is the vulnerability?**

In foo, target0 copies argv[1] into a buffer using strcpy without any bounds checking. This enables a buffer overflow in the case of arguments which are longer than the size of the buf.

**(2-3 sentences) What is the conceptual approach of the exploit?**

Provide an overly long string as input to target0. It needs to be at least 8 bytes longer than sizeof(buf), so that strcpy will overwrite the 8 bytes after buf (which are the saved EBP and saved EIP in foo's stackframe). It should also contain the shellcode, which will be copied into buf and executed from there. The last 4 bytes of the string (before the null byte) should be a pointer into buf where the shellcode will end up after being strcpy'd there.

**(Bulleted list of steps) What steps would you follow to implement the exploit again?**

* In sploit0.c:
  + Create an evil buffer of size sizeof(buf) + 8 + 1 (for the null byte). Fill the buffer with non-null bytes and null terminate it, so it is a C-string of the correct length.
  + Making the argument the right length will ensure that the memory addresses in target0 are the same as when we actually run the exploit.
  + Pass the evil buffer to target0 as its first argument.
* Run sploit0 in cgdb.
* In cgdb:
  + break foo
  + run
  + p &buf // Write down the output; call it ADDR\_BUF
  + exit
* In sploit0.c:
  + Make a #define ADDR\_BUF and set it to the value we wrote down earlier.
  + memcpy shellcode to the beginning of the evil argument buffer.
  + Make sure there's not a null byte after the shellcode in the evil argument buffer.
  + Write ADDR\_BUF to the last 4 bytes of the evil buffer before the null terminator, making sure to respect endianness.
  + Ensure that the space between the shellcode and ADDR\_BUF is filled with non-null bytes.
* Run sploit0.
* whoami
  + hax0red0

Jocelyn Shiue

2018/10/17

**Exploit 1**

**(1-2 sentences) What is the vulnerability?**

The vulnerability for exploit 1 is the same as exploit 0. In foo, target0 copies argv[1] into a buffer using strcpy without any bounds checking. This enables a buffer overflow in the case of arguments which are longer than the size of the buf.

**(2-3 sentences) What is the conceptual approach of the exploit?**

The conceptual approach for exploit 1 is the same as exploit 0. Provide an overly long string as input to target0. It needs to be at least 8 bytes longer than sizeof(buf), so that strcpy will overwrite the 8 bytes after buf (which are the saved EBP and saved EIP in foo's stackframe). It should also contain the shellcode, which will be copied into buf and executed from there. The last 4 bytes of the string (before the null byte) should be a pointer into buf where the shellcode will end up after being strcpy'd there.

**(Bulleted list of steps) What steps would you follow to implement the exploit again?**

* In sploit1.c:
  + Create an evil buffer of size sizeof(buf) + 8 + 1 (for the null byte). Fill the buffer with non-null bytes and null terminate it, so it is a C-string of the correct length.
  + Making the argument the right length will ensure that the memory addresses in target1 are the same as when we actually run the exploit.
  + Pass the evil buffer to target1 as its first argument.
* Run sploit0 in cgdb.
* In cgdb:
  + break foo
  + run
  + p &buf // Write down the output; call it ADDR\_BUF
  + exit
* In sploit1.c:
  + Make a #define ADDR\_BUF and set it to the value we wrote down earlier.
  + memcpy shellcode to the beginning of the evil argument buffer.
  + Make sure there's not a null byte after the shellcode in the evil argument buffer.
  + Write ADDR\_BUF to the last 4 bytes of the evil buffer before the null terminator, making sure to respect endianness.
  + Ensure that the space between the shellcode and ADDR\_BUF is filled with non-null bytes.
* Run sploit1.
* whoami
  + hax0red0

Jocelyn Shiue

2018/10/17

**Exploit 2**

**(1-2 sentences) What is the vulnerability?**

Since the for loop is written like this

  for (i = 0; i <= len; i++)

    out[i] = in[i];

There is one extra byte that we can overwrite, as 0 to length is actually (length + 1) values.

**(2-3 sentences) What is the conceptual approach of the exploit?**

We want to manipulate this extra byte to execute our malicious shellcode. In order to do so, we must trick the code in thinking that the address it should return to (eip) is the address to shellcode. We will utilize the byte that we can overwrite to point to 4 before where we store a pointer to buf, so that the code will interpret a junk ebp (the first 4 bytes) and our intended eip (the pointer to buf).

**(Bulleted list of steps) What steps would you follow to implement the exploit again?**

* In sploit2.c:
  + Create an evil buffer of size sizeof(buf) + 1 + 1 (for byte we can overwrite and the null byte). Fill the buffer with non-null bytes and null terminate it, so it is a C-string of the correct length.
  + Making the argument the right length will ensure that the memory addresses in target2 are the same as when we actually run the exploit.
  + Pass the evil buffer to target2 as its first argument.
* Run sploit2 in cgdb.
* In cgdb:
  + break foo
  + run
  + p &buf // Write down the output; call it ADDR\_BUF
  + exit
* In sploit1.c:
  + Make a #define ADDR\_BUF and set it to the value we wrote down earlier.
  + memcpy shellcode to the beginning of the evil argument buffer.
  + Make sure there's not a null byte after the shellcode in the evil argument buffer.
  + Insert a pointer that has the address of BUF, making sure to not overwrite BUF itself (recommended at location evil + 180)
  + Calculate the address 4 before that pointer, and take the last byte of the address. Write in that byte right before the null byte.
* Run sploit2.
* whoami
  + hax0red0

Jocelyn Shiue

2018/10/17

**Exploit 3**

**(1-2 sentences) What is the vulnerability?**

The vulnerability for exploit 3 is the same as exploit 2. Since the for loop is written like this

  for (i = 0; i <= len; i++)

    out[i] = in[i];

There is one extra byte that we can overwrite, as 0 to length is actually (length + 1) values.

Exploit 3 differs in that there is no ebp in this environment.

**(2-3 sentences) What is the conceptual approach of the exploit?**

We want to manipulate this extra byte to execute our malicious shellcode. In order to do so, we must trick the code in thinking that the address it should return to (eip) is the address to shellcode. As there is no ebp in this environment, we will utilize the byte that we can overwrite to point to a return instruction. Thus, this return instruction will look at the next argument in the stack (which we will write as address to shellcode) to execute.

**(Bulleted list of steps) What steps would you follow to implement the exploit again?**

* In sploit3.c:
  + Create an evil buffer of size sizeof(buf) + 1 + 1 (for byte we can overwrite and the null byte). Fill the buffer with non-null bytes and null terminate it, so it is a C-string of the correct length.
  + Making the argument the right length will ensure that the memory addresses in target3 are the same as when we actually run the exploit.
  + Pass the evil buffer to target3 as its first argument.
* Run sploit3 in cgdb.
* In cgdb:
  + break foo
  + run
  + p &buf // Write down the output; call it ADDR\_BUF
  + i r //examine the value of eip
  + Find a return statement you can overwrite the eip to, by looking around the potential values near given eip (with overwritten last byte)
    - x/40 addr\_eip\_with\_last\_byte\_as\_00 //should find that 04 is useful
  + exit
* In sploit1.c:
  + Make a #define ADDR\_BUF and set it to the value we wrote down earlier.
  + memcpy shellcode to the beginning of the evil argument buffer.
  + Make sure there's not a null byte after the shellcode in the evil argument buffer.
  + Set the byte we can trick at evil[sizeof(buf)] to be the byte that leads to a return instruction //should be ‘\x04’
* Run sploit3.
* whoami
  + hax0red0

Jocelyn Shiue

2018/10/18

**Exploit 4**

**(1-2 sentences) What is the vulnerability?**

The vulnerability for exploit 4 is the similar as exploit 0. In foo, target0 copies argv[1] into a buffer using strcpy with the parameter of a short. We can exploit this short, as a short only reads a limited number of bytes, passing in a number that is much larger than 496 but will be interpreted as less. This enables a buffer overflow in the case of arguments which are longer than the size of the buf.

**(2-3 sentences) What is the conceptual approach of the exploit?**

memset(evil, 0x90, 100);

Nop arbitrary amount of times before you hit

The conceptual approach for exploit 4 is similar as exploit 0. Provide an overly long string as input to target0. It needs to be at least 8 bytes longer than sizeof(buf), so that strcpy will overwrite the 8 bytes after buf (which are the saved EBP and saved EIP in foo's stackframe). It should also contain the shellcode, which will be copied into buf and executed from there. The location of eip should be overwritten by a pointer into buf where the shellcode will end up after being strcpy'd there.

**(Bulleted list of steps) What steps would you follow to implement the exploit again?**

* In sploit4.c:
  + Create an evil buffer of massive size that the short will interpret to be less than 496 //recommended 65930
  + Fill the buffer with non-null bytes and null terminate it
  + Pass the evil buffer to target1 as its first argument.
* Run sploit0 in cgdb.
* In cgdb:
  + break foo
  + run
  + p &buf // Write down the output; call it ADDR\_BUF
  + exit
* In sploit1.c:
  + Make a #define ADDR\_BUF and set it to the value we wrote down earlier.
  + memcpy shellcode to the beginning of the evil argument buffer.
  + Make sure there's not a null byte after the shellcode in the evil argument buffer.
  + At the location of eip (496 + 4 + 4) write a pointer to ADDR\_BUF
  + Ensure that the space between the shellcode and the pointer ADDR\_BUF is filled with non-null bytes.
* Run sploit4.
* whoami
  + hax0red0

Jocelyn Shiue, 2018/10/18

**Exploit 5**

**(1-2 sentences) What is the vulnerability?**

The vulnerability for exploit 5 is that tfree(second) is called a second time. This takes advantage of the fact that even though we should not, we have access to the area which second points to and can trick the system into going to a specified address.

**(2-3 sentences) What is the conceptual approach of the exploit?**

We want to trick the system into running shellcode. We do this by our buffer overflow and overwriting data. We will overwrite the chunk’s processes to input a pointer to the beginning of buf at seip. To do that, we must change the free bit in the location of second manually using evil. However, simply going to the beginning of buf is not enough. We must use an assembly jump to shellcode instruction at the beginning of evil to get to the shellcode.

**(Bulleted list of steps) What steps would you follow to implement the exploit again?**

* In sploit5.c:
  + Create an evil buffer of a size at least greater than 324. This is so that we can be certain to overwrite the location of the chunk which changes seip. Fill the buffer with non-null bytes and null terminate it, so it is a C-string of the correct length.
  + Pass the evil buffer to target1 as its first argument.
* Run sploit5 in cgdb.
* In cgdb:
  + break foo
  + run
  + p &buf // Write down the output; call it ADDR\_BUF
  + p &second //Write down the output; call it second’s address
  + I f //Write down the address of seip
  + exit
* In sploit5.c:
  + Make a #define ADDR\_BUF and set it to the value we wrote down earlier.
  + Memset(evil, 0x90, some\_number\_I\_recommend\_100). This is to write an arbitrary number of NOP which allows the program to keep jumping until it has reached runnable code
  + memcpy shellcode at an address some ways in of the evil argument buffer (recommended 30)
  + Add assembly jump instructions at the very beginning of buf memcpy(evil, “\xeb\x08”, 2)
  + Set the free bit of the chunk by making evil[4] = -1, meaning that the last bit will be odd for certain.
  + Overwrite the addresses which second will use to “tfree” the fake chunk
    - Set 4 before where second points to be the address of the start of buf \* (int \*) (evil + 324) = 0xffffdbe0
    - Set 8 before where second points to be the address of seip \* (int \*) (evil + 320) = 0x8049dc8
  + Make sure there's not a null byte after the shellcode in the evil argument buffer.
* Run sploit5.
* whoami
  + hax0red0

Jocelyn Shiue

2018/10/17

**Exploit 6**

**(1-2 sentences) What is the vulnerability?**

The vulnerability for exploit 6 is that the program utilizes the method snprintf. The method snprintf(buf, sizeof buf, arg) allows a string to be executed as a command. Our end objective is to execute the malicious shell code. To do so, one needs to write an address to the shellcode as an integer and execute it from the snprintf statement.

**(2-3 sentences) What is the conceptual approach of the exploit?**

The conceptual approach for exploit 6 is to trick snprintf into executing the shellcode. In order to have snprintf read the address of shellcode, we will utilize %n and %d to overwrite the foo’s eip. To do so, we must overwrite eip in increments of bytes. The address of eip should be listed right after buf after being sandwiched between 4 junk bytes. These junk bytes are so that nextarg will properly read the address. Following that, another four junk bytes should be written and the address one directly after seip. Continue this pattern until you have completed the four addresses after eip. The shellcode should follow directly afterwards, and after that we strategically enter numbers between %d and %n to summate to the address of buf.

**(Bulleted list of steps) What steps would you follow to implement the exploit again?**

* In sploit6.c:
  + Create an evil buffer of size sizeof(buf) + 8 + 1 (for the null byte). Fill the buffer with non-null bytes and null terminate it, so it is a C-string of the correct length.
  + Making the argument the right length will ensure that the memory addresses in target1 are the same as when we actually run the exploit.
  + Pass the evil buffer to target1 as its first argument.
* Run sploit6 in cgdb.
* In cgdb:
  + break foo
  + run
  + p &buf // Write down the output; call it ADDR\_BUF
  + i r //Write down the address of eip
  + exit
* In sploit6.c:
  + Make a #define ADDR\_BUF and set it to the value we wrote down earlier.
  + Memset a filler character to the size of evil
  + Memcpy 4 filler bytes followed by the address to eip, 4 filler bytes followed by the address to eip + 1… etc until you get to eip + 3 to the beginning of the evil argument buffer.
  + Memcpy shellcode after the addresses (should be at evil + 28)
  + Make sure there's not a null byte after the shellcode in the evil argument buffer.
  + Memcpy the %d and %n which add up to the address of where shellcode is located in the buffer (ADDR\_BUF + 28).
* Run sploit6.
* whoami
  + hax0red0

Jocelyn Shiue

2018/10/22

**Exploit 7**

**(1-2 sentences) What is the vulnerability?**

Similar to sploit 2 and 3, the poorly written for loop within nstrcpy() allows for us to change one byte of the memory address pointer which the program is returning to. We want to manipulate this byte to change the frame so that we can execute the shellcode which will also involve the pointer vulnerability within foo. The line \*p = a; means that the pointer is getting the contents (not the address) of a, which is earlier declared as 0.

**(2-3 sentences) What is the conceptual approach of the exploit?**

Firstly, we will use that byte that we can use to overwrite to trick the code into thinking it is in a different ebp. This will allow us to utilize evil to overwrite the addresses contained in p and a so that a contains the address of buf, and p points to an exit instruction. After the exit instruction, the code will return and execute shellcode stored in a. To make these pointers execute in the program we will use the one byte that we have access to and overwrite it to the location of p.

**(Bulleted list of steps) What steps would you follow to implement the exploit again?**

* In sploit7.c:
  + Create an evil buffer that is long enough to overwrite the one byte and add in the null byte. This should be 184+1+1. Fill the buffer with non-null bytes and null terminate it, so it is a C-string of the correct length.
  + Pass the evil buffer to target1 as its first argument.
* Run sploit7 in cgdb.
* In cgdb:
  + break foo
  + run
  + p &buf // Write down the output; call it ADDR\_BUF
  + I f //Look for ebp, write it down
  + p &p //examine the locations of pointer p
  + p &a //examine the location of a
  + exit
  + disas \_exit
  + look for a jmp instruction that we can manipulate, utilizing the one which is writeable. //Write down the address of the jmp
* In sploit7.c:
  + Make a #define ADDR\_BUF and set it to the value we wrote down earlier.
  + memcpy shellcode to the beginning of the evil argument buffer.
  + Make sure there's not a null byte after the shellcode in the evil argument buffer.
  + Overwrite the byte to be the ebp of the previous block so that p and a can get updated
  + Change the values within p and a so that p contains the address of the jmp instruction within \_exit, and p to be the address of the malicious shellcode
* Run sploit1.
* whoami
  + hax0red0

Jocelyn Shiue

2018/10/22

**Exploit 7**

**(1-2 sentences) What is the vulnerability?**

Similar to sploit 2 and 3, the poorly written for loop within nstrcpy() allows for us to change one byte of the memory address pointer which the program is returning to. We want to manipulate this byte to change the frame so that we can execute the shellcode which will also involve the pointer vulnerability within foo. The line \*p = a; means that the pointer is getting the contents (not the address) of a, which is earlier declared as 0.

**(2-3 sentences) What is the conceptual approach of the exploit?**

Firstly, we will use that byte that we can use to overwrite to trick the code into thinking it is in a different ebp. This will allow us to utilize evil to overwrite the addresses contained in p and a so that a contains the address of buf, and p points to an exit instruction. After the exit instruction, the code will return and execute shellcode stored in a. To make these pointers execute in the program we will use the one byte that we have access to and overwrite it to the location of p.

**(Bulleted list of steps) What steps would you follow to implement the exploit again?**

* In sploit7.c:
  + Create an evil buffer that is long enough to overwrite the one byte and add in the null byte. This should be 184+1+1. Fill the buffer with non-null bytes and null terminate it, so it is a C-string of the correct length.
  + Pass the evil buffer to target1 as its first argument.
* Run sploit7 in cgdb.
* In cgdb:
  + break foo
  + run
  + p &buf // Write down the output; call it ADDR\_BUF
  + I f //Look for ebp, write it down
  + p &p //examine the locations of pointer p
  + p &a //examine the location of a
  + exit
  + disas \_exit
  + look for a jmp instruction that we can manipulate, utilizing the one which is writeable. //Write down the address of the jmp
* In sploit7.c:
  + Make a #define ADDR\_BUF and set it to the value we wrote down earlier.
  + memcpy shellcode to the beginning of the evil argument buffer.
  + Make sure there's not a null byte after the shellcode in the evil argument buffer.
  + Overwrite the byte to be the ebp of the previous block so that p and a can get updated
  + Change the values within p and a so that p contains the address of the jmp instruction within \_exit, and p to be the address of the malicious shellcode
* Run sploit1.
* whoami
  + hax0red0

Jocelyn Shiue

2018/10/22

**Exploit 8**

**(1-2 sentences) What is the vulnerability?**

The vulnerability is that for strcpy, there is no limit to amount you can copy over

strcpy(buf, argv[1]);

However, the parameters for the environment were set such that only code in the text region would execute. Because of this we will need to utilize existing code for this to successfully run.

**(2-3 sentences) What is the conceptual approach of the exploit?**

The main goal of this exploit is to execute “bin/sh” so that a shell will appear. To do this, we will need to place a pointer to bin within eax so that it will be executed. If you disas system, you will find that there is a location within the region of buf where something is moved into eax. We will overwrite this as a pointer to bin. Additionally, we will set the string “bin/sh” to be 4 addresses after the pointer as this is the location where the arguments lie.

**(Bulleted list of steps) What steps would you follow to implement the exploit again?**

* In sploit8.c:
  + Create an evil buffer that is long enough to overwrite the one byte and add in the null byte. This should be 343+1.
  + Fill the buffer with non-null bytes and null terminate it, so it is a C-string of the correct length.
* Run sploit8 in cgdb.
* In cgdb:
  + break foo
  + run
  + p &buf // Write down the output; call it ADDR\_BUF
  + disas system //Write down the address of the first instruction
  + observe the location of esp as the program moves esp + 0x10 into eax
  + I f //observe the location of esp
  + exit
* In sploit8.c:
  + Choose a location to input the address of system within buf. (Recommended evil + 324).
  + Note the difference between the address of esp + 0x10 (eax) and buf is 332, so it is there that we will put a pointer to bin.
  + The pointer of bin should be the address of where we put the address of system. (In the situation where you put the address of system at evil + 324, the pointer to bin should contain the address 0xffffdd1c)
  + Four after the location we put the pointer to bin, put the arguments. Since we want to run shell code, we will memcpy “/bin/sh” at evil + 336.
* Run sploit8.
* whoami
  + hax0red0