A Scaffolded, Metamorphic CTF for Reverse Engineering

Wu-chang Feng
Portland State University
Department of Computer Science

Abstract

Hands-on Capture-the-Flag (CTF) challenges tap into and cultivate the intrinsic motivation within people to solve puzzles, much in the same way Sudoku and crossword puzzles do. While the format has been successful in security competitions, there have been a limited number of attempts to integrate them into a classroom environment. This paper describes MetaCTF, a metamorphic set of CTF challenges for teaching reverse code engineering. MetaCTF is 1) scaffolded in a way that allows students to make incremental progress, 2) integrated with the course material so that students can immediately apply knowledge gained in class, 3) polymorphic and metamorphic so that individual students within a class and between multiple offerings of a class are given unique challenges, and 4) extensible in order to allow students to design their own CTF challenges that can be later integrated into future offerings of the course.

1 Introduction

Because the goal of CTF competitions is to evaluate competency of practitioners rather than teach them new skills, such events often use highly challenging problems that are intended to measure skills practitioners have already acquired and mastered. As a result, the challenges used are often open-ended and unguided, which can lead to frustration for beginners rather than the steady learning, skill development, and confidence building that one would like to cultivate in courses [1, 2]. Further, in order to determine a "winner", many challenges used in competitive CTF events are only meant to be finished by a select few and can thus lead to designs that are intentionally esoteric with limited pedagogy. Such designs force individuals on teams to specialize on specific tasks rather than apply themselves to all tasks.

There have been promising initial efforts for adapting CTFs in a way that is suitable for teaching curricula [3]. Such efforts focus on skill development rather than skill evaluation. In addition, several CTFs are now targeting beginners [4, 5, 6, 7] and many on-line CTF sites have developed scaffolded challenges that help incrementally bring users to a level of competency one step at a time [8, 9, 10]. Inspired by these efforts, MetaCTF targets the construction of CTF exercises for use in teaching courses. MetaCTF challenges are tied to specific course objectives and are scaffolded in a way to allow all students to quickly progress along a path towards mastery. This paper describes our initial CTF for teaching reverse-engineering and malware analysis [11]. The course is based on the book "Practical Malware Analysis" by Sikorski and Honig [12].

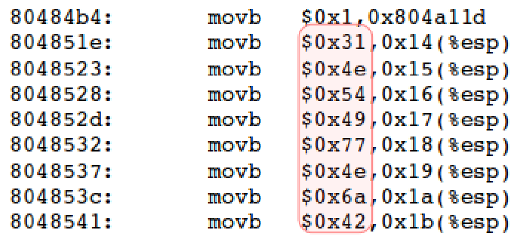
2 General Approach

While a more open-ended format is typical in CTF design, such a format has been shown to lead to poorer outcomes when applied to teaching new skills to beginners [13]. As a result, MetaCTF adopts a direct instruction approach in its challenges [14] with the understanding that the skills acquired through these challenges would then allow students to quickly transition towards formats more compatible to discovery learning [15]. Towards this end, MetaCTF intentionally uses the same general design across all levels to maximize the student's focus on specific technical topics. It also provides instructional material to point students in the right direction, often referring to content that has been supplied previously in the textbook. While each level increases in sophistication as students progress, the intent is for each level to be solvable in under an hour once students have been given the lecture associated with the level. This results in level designs that focus narrowly on the chapter topics.

The CTF levels themselves all mirror Lab 15-1 in the textbook [12] and also borrow concepts found in Ptacek et. al.'s microCorruption CTF in which a single `unlock_door` function, or code that is its equivalent, is repeatedly targeted for execution [10, 16] Each level prompts the user for a password that unlocks the binary and causes it to print the string "Good Job.". The CTF begins with levels that can be solved easily via simple static and dynamic analysis, but quickly progresses until the final levels in which students must bypass advanced anti-debugging and packing methods to solve the challenges. Because the examples, labs, and software tools that the textbook uses are all Windows-based, the MetaCTF challenges are all Linux binaries in order to force students to apply the same concepts and methodology using an alternate environment and set of tools. Finally, because challenges can be solved in many different ways, several challenges employ more sophisticated techniques that have not yet been introduced to students in order to encourage students to solve them in the intended manner.

3 Level Designs

MetaCTF aims to scaffold challenges in a way that those who undertake it can quickly progress towards competency in reverse-engineering. Maximizing the number of people who can complete all challenges while minimizing the amount of time they need to get there, requires carefully designed challenges that are focused in a way that incrementally builds skill and confidence in those solving them. Consider the two most difficult reverse-engineering challenges from the initial version of picoCTF [6]: `Mildly Evil` and `moreevil`. Both binaries use advanced techniques in anti-disassembly and anti-debugging that are not commonly known, resulting in a tiny fraction of teams attempting them, and out of those, an even tinier fraction of teams solving them. In the context of a competition, such binaries are ideal since they provide clear separation between the top percentile of teams and the rest of teams. The goal of MetaCTF, however, is the opposite. Its challenges are intended to maximize the number of students who can reach the level required to solve these difficult challenges.



```
80484b4:        movb   $0x1,0x804a11d
804851e:        movb   $0x31,0x14(%esp)
8048523:        movb   $0x4e,0x15(%esp)
8048528:        movb   $0x54,0x16(%esp)
804852d:        movb   $0x49,0x17(%esp)
8048532:        movb   $0x77,0x18(%esp)
8048537:        movb   $0x4e,0x19(%esp)
804853c:        movb   $0x6a,0x1a(%esp)
8048541:        movb   $0x42,0x1b(%esp)
```

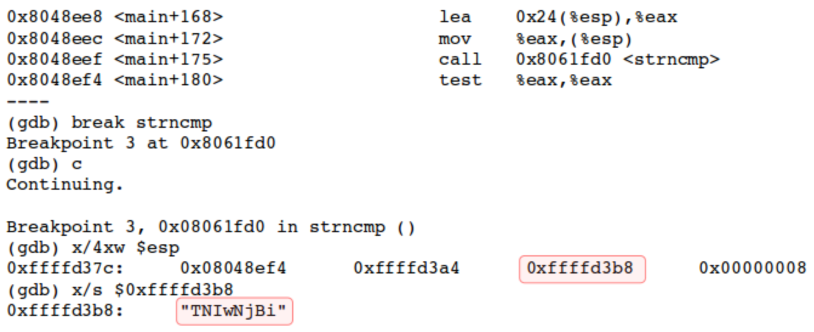
Figure 1: Password embedded in assembly

3.1 Basic static and dynamic analysis

The challenges begin with basic static analysis in which students apply binary analysis tools to find the appropriate section that contains the password. While the lab assignments in the textbook walk students through the use of Windows tools such as `PEView`, the challenges compel students to apply Linux equivalents such as `readelf` and `objdump` to discover the password that is embedded in the clear within a binary's read-only data section. It is easy to circumvent static analysis techniques by dynamically generating the password. While in the textbook's lab assignments, students apply tools such as `Process Explorer`, `Process Monitor`, and `regshot`, the challenges instead have students familiarize themselves with basic dynamic tracing tools on Linux such as `ltrace` and `strace`. Through the use of simple dynamic tracing, the library calls that the program uses are captured. The password is then observed as it is being checked via a library call to `strcmp`.

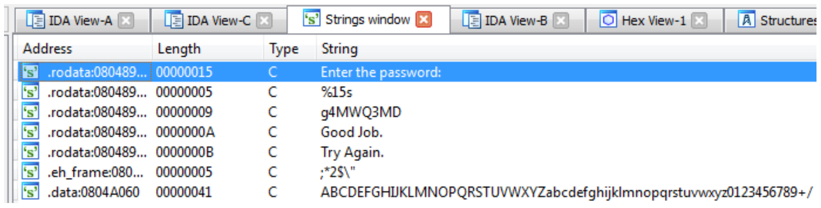
3.2 Disassembling and debugging

Most reverse-engineering challenges require the use of a disassembler and a debugger. In the lab assignments, students use standard tools such as `OllyDbg` and `IDA Pro` to analyze malicious assembly code. To complement this, the next set of challenges force students to use `objdump` and `gdb` to accomplish the same on Linux. In one challenge, the password is hard-coded within the instructions themselves. As Figure 1 shows, solving this challenge requires students to look for the embedded ASCII characters within instructions in the binary's text section. In another challenge, the password is generated at run-time before a call to `strcmp`. To prevent library tracing attacks, the challenge is statically compiled. In order to solve this challenge, students are required to apply their knowledge of `gdb` and stack frames. They must first identify where the call to `strcmp` is located, set a breakpoint where the compare occurs, and then identify



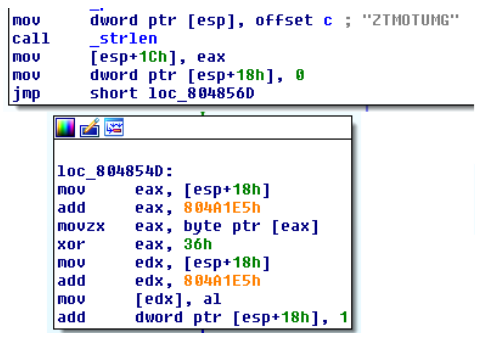
```
0x8048ee8 <main+168>                    lea    0x24(%esp),%eax
0x8048eec <main+172>                    mov    %eax,(%esp)
0x8048eef <main+175>                    call   0x8061fd0 <strncmp>
0x8048ef4 <main+180>                    test   %eax,%eax
----
(gdb) break strncmp
Breakpoint 3 at 0x8061fd0
(gdb) c
Continuing.

Breakpoint 3, 0x08061fd0 in strncmp ()
(gdb) x/4xw $esp
0xffffd37c:     0x08048ef4      0xffffd3a4      0xffffd3b8      0x00000008
(gdb) x/s $0xffffd3b8
0xffffd3b8:     "TNIwNjBi"
```

Figure 2: Password captured within debugger



Address	Length	Type	String
.rodata:080489...	00000015	C	Enter the password:
.rodata:080489...	00000005	C	%15s
.rodata:080489...	00000009	C	g4MWQ3MD
.rodata:080489...	0000000A	C	Good Job.
.rodata:080489...	0000000B	C	Try Again.
.eh_frame:080...	00000005	C	;*2$\"
.data:0804A060	00000041	C	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/

Figure 3: Base64 challenge



```
mov     dword ptr [esp], offset c ; "ZTMOTUMG"
call    _strlen
mov     [esp+1Ch], eax
mov     dword ptr [esp+18h], 0
jmp     short loc_804856D
```

```
loc_804854D:
mov     eax, [esp+18h]
add     eax, 804A1E5h
movzx   eax, byte ptr [eax]
xor     eax, 36h
mov     edx, [esp+18h]
add     edx, 804A1E5h
mov     [edx], al
add     dword ptr [esp+18h], 1
```

Figure 4: XOR challenge

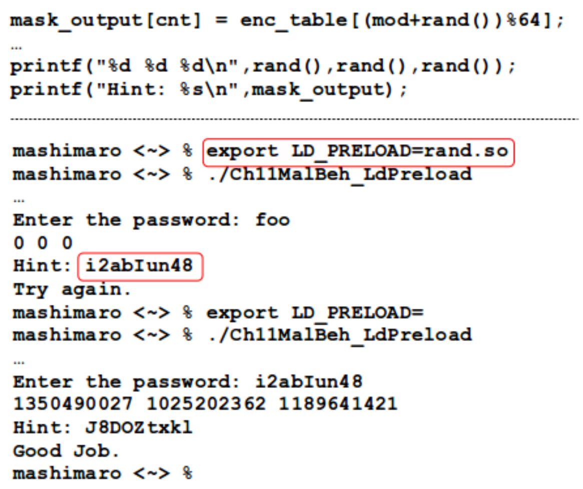
the arguments to the call at that time. Figure 2 shows how a student would use the debugger to stop the program within the statically linked `strcmp` call in order to reveal the password passed as one of the function parameters.

3.3 Data encoding

Malware often encodes its payloads in order to thwart signature-detection schemes. In the next set of challenges, students are forced to identify encoding algorithms used in assembly so that they can then reverse them. Figure 3 shows one of the challenges adapted from the textbook in which students must identify that a Base64 encoding format is being used to generate the password. Programs that use Base64 often store an array consisting of all of the characters in the encoding so that conversion to and from the format can be done via an indexing operation on the array. When the challenge is brought up in a disassembler, students can examine the strings embedded in the binary and find the Base64 character array. They can then reverse the encoding to recover the actual password. Figure 4 shows a similar example in which the password is XOR encoded using a constant byte. In this challenge, students are required to examine the assembly code of the binary to see individual characters of the password string being XOR'd in a tight loop. They are then able to reverse the operation manually or set appropriate breakpoints within the debugger to obtain the password.

3.4 Malware behavior

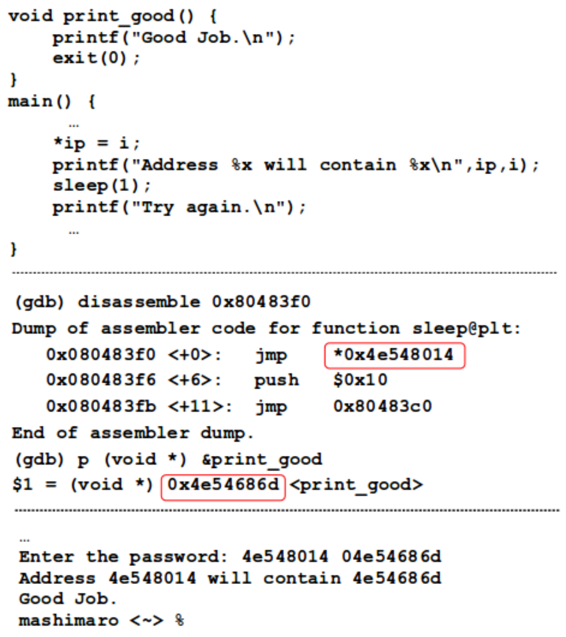
Once the preliminaries are out of the way, students can now focus on how malware interacts with the underlying system it is attempting to compromise. In the lab assignments, students explore how malware hijacks normal execution on a Windows machine by tampering with the registry, binaries, library paths, and running processes. The challenges attempt to mirror this by forcing students to hijack execution of the binary in order to solve the challenges. Figure 5 shows a challenge in which the password is generated dynamically via an iterative mathematical operation that is intended to be somewhat dif-



```
mask_output[cnt] = enc_table[(mod+rand())%64];
...
printf("%d %d %d\n",rand(),rand(),rand());
printf("Hint: %s\n",mask_output);
```

```
mashimaro <~> % export LD_PRELOAD=rand.so
mashimaro <~> % ./Ch11MalBeh_LdPreload
...
Enter the password: foo
0 0 0
Hint: i2abIun48
Try again.
mashimaro <~> % export LD_PRELOAD=
mashimaro <~> % ./Ch11MalBeh_LdPreload
...
Enter the password: i2abIun48
1350490027 1025202362 1189641421
Hint: J8DOZtxkl
Good Job.
mashimaro <~> %
```

Figure 5: Library hijacking challenge



```
void print_good() {
    printf("Good Job.\n");
    exit(0);
}
main() {
     ...
    *ip = i;
    printf("Address %x will contain %x\n",ip,i);
    sleep(1);
    printf("Try again.\n");
     ...
}
```

```
(gdb) disassemble 0x80483f0
Dump of assembler code for function sleep@plt:
   0x080483f0 <+0>:    jmp    *0x4e548014
   0x080483f6 <+6>:    push   $0x10
   0x080483fb <+11>:   jmp    0x80483c0
End of assembler dump.
(gdb) p (void *) &print_good
$1 = (void *) 0x4e54686d <print_good>
```

```
...
Enter the password: 4e548014 04e54686d
Address 4e548014 will contain 4e54686d
Good Job.
mashimaro <~> %
```

Figure 6: Procedure Link Table hijacking

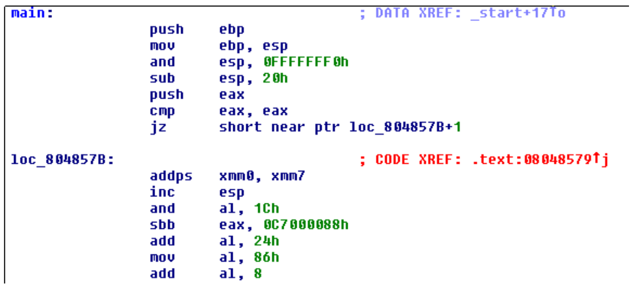
ficult to reverse. Note that the goal is to make it more difficult to reverse the computation than to hijack execution. While the password is being computed, a hint is generated in which the password is modified using calls to libc's `rand()` function. The goal is to create a rogue library that hijacks the call to `rand()` so that it always returns 0. In this case, the hint ends up being the password. Another method for hijacking execution shown in the lab assignments is to attack the import address table (IAT) of a Windows binary. This table supports the level of indirection required to implement dynamically linked libraries whose load addresses are not known until run-time. Adapted from the Vortex OverTheWire challenges [9], Figure 6 shows a challenge in which students learn how memory corruption errors can be used to hijack execution via the targeted overwriting of the Linux-equivalent to the IAT: the procedure link table (PLT). In this challenge, students are given a single memory write to an arbitrary address that will cause the binary to unlock itself. Using `objdump` and `gdb`, students must identify which function call to hijack, where that function call's PLT entry is, and the address of the function they wish to call instead.

3.5 Anti-disassembly

At this point in the course, students are assumed to be proficient in binary analysis and reversing using basic static and dynamic analysis. As malware has evolved, counter-measures against such analysis are often deployed. Subsequent levels of the CTF allow students to incrementally learn how to bypass each one. One counter-measure that is employed is to confuse static analysis of malware's assembly code via garbage instructions [17]. Figure 7(a) shows an example of a fake conditional jump that confuses binary disassemblers. In this case, the program compares a register to itself and "conditionally" branches if they are equal. The branch is followed by a random byte which causes sequential disassemblers to produce unregonizable output. For this challenge, we disable the use of debuggers using a `ptrace` check on program startup to force students to bypass the problematic conditional jump in order to solve the level. This design highlights the scaffolding in the challenges where a technique from a subsequent chapter (i.e. anti-debugging) is used to ensure that students solve a challenge in a particular way (i.e. via the use of a disassembler). In this particular challenge, students must identify the location of the problematic conditional jump and force the disassembler to reinterpret the opcodes to reflect the actual execution path of the program. Figure 7(b) shows the same code after students bypass the anti-disassembly technique. Additional anti-disassembly challenges for this chapter force students to identify and bypass fake call instructions, inward jumping instructions, obfuscated jump instructions, and impossible disassembly techniques.

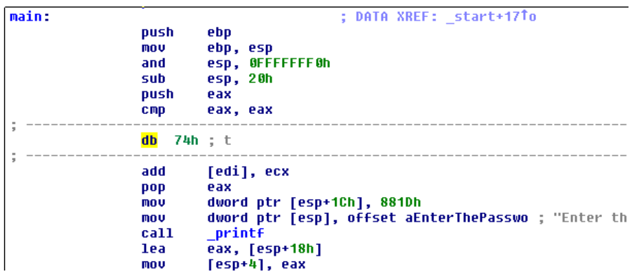
3.6 Anti-debugging

To prevent analysis, modern malware employs a variety of methods to thwart debuggers. In the lab as-



```
main:                                         ; DATA XREF: _start+17↑o
                    push    ebp
                    mov     ebp, esp
                    and     esp, 0FFFFFFF0h
                    sub     esp, 20h
                    push    eax
                    cmp     eax, eax
                    jz      short near ptr loc_804857B+1

loc_804857B:                                  ; CODE XREF: .text:08048579↑j
                    addps   xmm0, xmm7
                    inc     esp
                    and     al, 1Ch
                    sbb     eax, 0C7000088h
                    add     al, 24h
                    mov     al, 86h
                    add     al, 8
```

(a) Fake conditional guarding password code



```
main:                                         ; DATA XREF: _start+17↑o
                    push    ebp
                    mov     ebp, esp
                    and     esp, 0FFFFFFF0h
                    sub     esp, 20h
                    push    eax
                    cmp     eax, eax
; ---------------------------------------------------------------------------
                    db  74h ; t
; ---------------------------------------------------------------------------
                    add     [edi], ecx
                    pop     eax
                    mov     dword ptr [esp+1Ch], 881Dh
                    mov     dword ptr [esp], offset aEnterThePasswo ; "Enter th
                    call    _printf
                    lea     eax, [esp+18h]
                    mov     [esp+4], eax
```

(b) Fake conditional bypassed

Figure 7: Anti-disassembly CTF challenge

signments, students learn how to bypass such methods within `OllyDbg` by identifying where anti-debugging code is being executed and ensuring that the artifacts being checked do not expose the presence of the debugger. In the CTF challenges for anti-debugging, the binaries either exit in the presence of a debugger or entangle their password in a way that produces an incorrect result when run via a debugger. Figure 8 shows two anti-debuggin methods that students need to identify and bypass in order to obtain the correct password. The first detects that a debugger has hijacked the SIGTRAP signal by putting the password code in the SIGTRAP handler. The exercise forces students to understand exception handling in the debugger. The second technique scans the program's code pages looking for places where the debugger has inserted a software interrupt instruction to handle regular breakpoints (0xcc) and changes the password if it detects interrupt insertions. To bypass this technique, the student would either need to use hardware breakpoints or reverse-engineer the scanning code in order to bypass it. Additional anti-debugging levels include the use of `ptrace` and `gdb` artifacts within the process, checking for the presence of a debugger-installed SIGTRAP handler, and checking for timing anomalies indicating debugger operation.

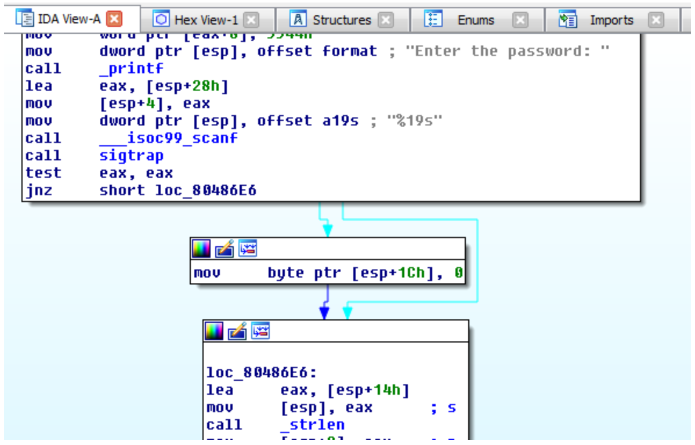
3.7 Packers and unpacking

Modern packers such as Themida [18] compress, encrypt, and obfuscate payloads to make analysis extremely difficult. While sophisticated packers might only decrypt one instruction at a time, others will unpack the entire payload into memory before transferring control over to the unpacked code. In the lab assignments, students are introduced to the function of simple packers and the unpacking operation. By tracing through the execution of a binary as it unpacks itself, they learn to identify where control is transferred from the unpacking code over to the original code. Similar to some of the labs in the textbook, in the CTF challenges for packers, students are tasked with unpacking a polymorphic version of an earlier binary that that has been packed by UPX [19]. In the first challenge they can simply unpack the binary using UPX. In a subsequent level, the packed binary is modified in a way that UPX is unable to unpack it, thus forcing a manual unpacking. In this challenge, students must use their knowledge of how unpackers work and employ dynamic run-time tracing to attach to the running code after it has been unpacked. They can then identify where the password is being checked and obtain its value.

4 Level Morphing

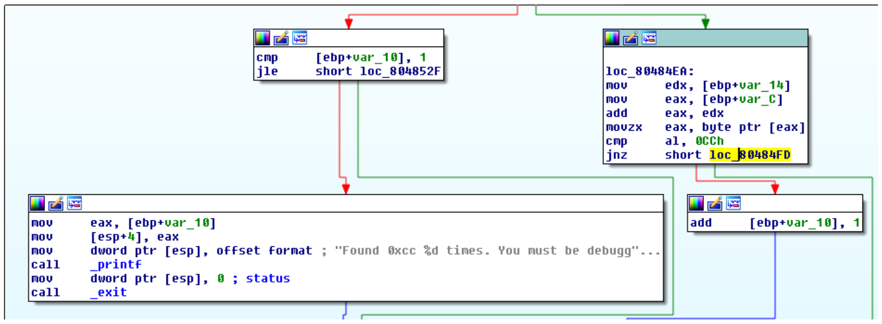
The scaffolding of challenges to match course objectives allows students to incrementally build competence and confidence as they learn the material. Because it is essential that each student goes through the process of developing these problem solving skills, CTF challenges must be designed to mitigate the opportunities for cheating. Such an issue is largely unaddressed in a competitive CTF event since there is no incentive for participants to share answers with each other. Thus, even though the challenges given to teams are all the same, each team typically tackles them individually.

While the existing model for CTFs is fine for competitions, it is a poor fit when used in the context of courses for several reasons. One reason is that unlike competitive CTF events, students do not have a strong incentive to keep their answers to themselves. Thus, there is no natural way to force students to work through challenges on their own. Another reason is that because courses are offered repeatedly, unless challenges are rewritten every time a course is offered, a student would only need to obtain an answer key to a previous offering in order to



```
mov     dword ptr [esp], offset format ; "Enter the password: "
call    _printf
lea     eax, [esp+28h]
mov     [esp+4], eax
mov     dword ptr [esp], offset a19s ; "%19s"
call    ___isoc99_scanf
call    sigtrap
test    eax, eax
jnz     short loc_80486E6

mov     byte ptr [esp+1Ch], 0

loc_80486E6:
lea     eax, [esp+14h]
mov     [esp], eax      ; s
call    _strlen
```

(a) Test SIGTRAP



```
cmp     [ebp+var_10], 1
jle     short loc_804852F

loc_80484EA:
mov     edx, [ebp+var_14]
mov     eax, [ebp+var_C]
add     eax, edx
movzx   eax, byte ptr [eax]
cmp     al, 0CCh
jnz     short loc_80484FD

mov     eax, [ebp+var_10]
mov     [esp+4], eax
mov     dword ptr [esp], offset format ; "Found 0xcc %d times. You must be debugg"...
call    _printf
mov     dword ptr [esp], 0 ; status
call    _exit

add     [ebp+var_10], 1
```

(b) Scan for INT 3 (0xcc)

Figure 8: Anti-debugging CTF challenge

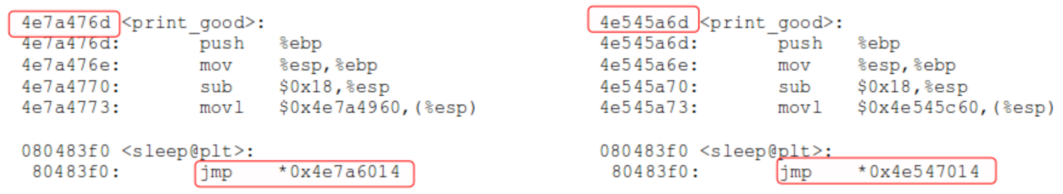
submit correct answers to the challenges without having to do any of the work or learning.

To address these issues, several CTFs [4, 10] employ polymorphism and metamorphism so that the challenges users solve are unique. Towards this end, all MetaCTF challenges are built to be either polymorphic or metamorphic in order to ensure that challenges are unique across students in a course and between students across multiple offerings of the course. While walkthroughs of the challenges might give a student an idea of how to solve them, they must still apply the concepts of the walkthrough on their specific challenge to solve it. By doing so, they automatically develop the skills and knowledge the CTF challenge wants of them.

MetaCTF employs a variety of techniques to generate polymorphic and metamorphic binaries. The simplest is to modify the program's data either as it is stored statically in the binary or as it is encoded in the instructions as described in the initial levels. Additional techniques employed by MetaCTF to ensure the uniqueness of each challenge include morphing the data structures used and the addresses where code and data are placed. Figure 9 shows two versions of the PLT hijacking challenge described previously. In this case, the addresses of the text section are randomly generated to force students to individually determine the values required to unlock the level. Finally, one of the most powerful ways to apply metamorphism to CTF challenges is to modify the code itself. Figure 10 shows an example where metamorphic code injection is used to create unique CTF challenges. In this case, the set of anti-disassembly techniques that are used to hide instructions in the binary are uniquely generated for each student. To solve this challenge, students must individually apply counter-measures to bypass their set of anti-disassembly techniques. Since the instructions themselves differ, even code-walkthroughs become more difficult to apply and follow.

5 Level Extensions

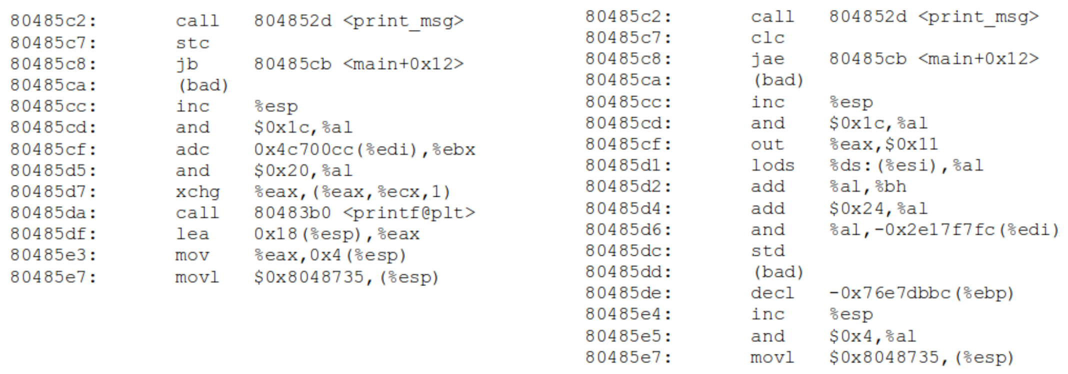
"See one, Do one, Teach one" is a medical school adage that is followed to help students obtain mastery. The lecture material and the MetaCTF challenges provide students with the first two. As part of our offering of the



```
4e7a476d <print_good>:
 4e7a476d:        push   %ebp
 4e7a476e:        mov    %esp,%ebp
 4e7a4770:        sub    $0x18,%esp
 4e7a4773:        movl   $0x4e7a4960,(%esp)

080483f0 <sleep@plt>:
 80483f0:         jmp    *0x4e7a6014
```

```
4e545a6d <print_good>:
 4e545a6d:        push   %ebp
 4e545a6e:        mov    %esp,%ebp
 4e545a70:        sub    $0x18,%esp
 4e545a73:        movl   $0x4e545c60,(%esp)

080483f0 <sleep@plt>:
 80483f0:         jmp    *0x4e547014
```

Figure 9: Metamorphic versions of a PLT corruption challenge



```
80485c2:        call   804852d <print_msg>
80485c7:        stc
80485c8:        jb     80485cb <main+0x12>
80485ca:        (bad)
80485cc:        inc    %esp
80485cd:        and    $0x1c,%al
80485cf:        adc    0x4c700cc(%edi),%ebx
80485d5:        and    $0x20,%al
80485d7:        xchg   %eax,(%eax,%ecx,1)
80485da:        call   80483b0 <printf@plt>
80485df:        lea    0x18(%esp),%eax
80485e3:        mov    %eax,0x4(%esp)
80485e7:        movl   $0x8048735,(%esp)
```

```
80485c2:        call   804852d <print_msg>
80485c7:        clc
80485c8:        jae    80485cb <main+0x12>
80485ca:        (bad)
80485cc:        inc    %esp
80485cd:        and    $0x1c,%al
80485cf:        out    %eax,$0x11
80485d1:        lods   %ds:(%esi),%al
80485d2:        add    %al,%bh
80485d4:        add    $0x24,%al
80485d6:        and    %al,-0x2e17f7fc(%edi)
80485dc:        std
80485dd:        (bad)
80485de:        decl   -0x76e7dbbc(%ebp)
80485e4:        inc    %esp
80485e5:        and    $0x4,%al
80485e7:        movl   $0x8048735,(%esp)
```

Figure 10: Metamorphic versions of an anti-disassembly CTF challenge

course, after completing the CTF challenges, students are tasked with the third. Specifically, they are assigned a final project in which they must develop their own metamorphic challenges that can be used to help someone learn a topic covered in the course. Students are encouraged to work on projects that fix scaffolding problems in the existing set of challenges.

MetaCTF provides a simple, straight-forward mechanism for students to add their own challenges. To create a custom challenge, a directory with two files is required. One file is a build script that takes a list of e-mail addresses as an argument. The other file is a source-code template for the challenge. Based on the e-mail address of a user, the build script generates random data that is then applied to the source-code template to generate a program that is polymorphic or metamorphic. In addition, the build script can optionally use the random data to modify where the text and data sections are preferentially loaded into memory. To make the task simpler, students are given sample source code for several of the challenges. In the latest offering of the course, students developed over a dozen reverse-engineering challenges. Several of them have been adapted and integrated into our current set of challenges.

6 Deployment

It can be cumbersome to run a CTF for a course. Instructors often don't have the time required to invest to adopt such an approach. To ease this burden, MetaCTF includes a web interface that can be used to manage the challenges. After the instructor configures a list of student e-mail addresses and builds the metamorphic binaries for each student, the web interface then is used to distribute individual challenges to students. It also is used to allow students to submit their solutions. Specifically, after finding solutions to binaries, users submit winning inputs to the web interface which then runs them in a sandbox against the binaries stored on the site to validate correctness. By providing an all-in-one solution for building, distributing, and grading CTF challenges, we hope to enable more widespread adoption of our approach.

7 Results

Consisting of an initial set of 17 metamorphic challenges, our first offering of our malware course in which the homeworks were given in the CTF format occurred during Winter quarter of 2015. Over the first 8 weeks of the course, students covered each chapter of the textbook using lectures, labs, and the CTF challenges associated

Term offered	Mean rating
Spring 2010	4.06
Winter 2011	4.11
Winter 2012	3.67
Winter 2013	4.25
Winter 2014	4.20
Winter 2015	4.67

Table 1: Quality and Usefulness of Homework Assignments for Malware course (1=Poor to 5=Very Good)

with the chapter. During the last 2 weeks, students then worked on developing their own challenges.

To evaluate differenct aspects of the course and the instructor, students are given surveys to fill out at the end of each course. Of particular interest in this study is the question that asks students to rate the quality and usefulness of the homework assignments. Table 1 shows the results pertaining to this question over the last 6 offerings of the malware course. As the table shows, while students felt that the homework quality and usefulness was good prior to the deployment of the CTF challenges, the mean rating of the homeworks significantly increased when the CTF was used.

8 Acknowledgments

Several students have contributed challenge designs including Simon Niklaus, Steven Tran, Tien Le, Yu Yang, and Stacy Watts. Work on MetaCTF has been supported by the Oregon Engineering and Technology Industry Council and the National Science Foundation under Grant No. CNS-1017034.

References

[1] K. Chung and J. Cohen, "Learning Obstacles in the Capture The Flag Model," in *USENIX 3GSE*, August 2014.

[2] G. Vigna, K. Borgolte, J. Corbetta, A. Doupe, Y. Fratantonio, L. Invernizzi, D. Kirat, and Y. Shoshitaishvili, "Ten Years of iCTF: The Good, The Bad, and The Ugly," in *USENIX 3GSE*, August 2014.

[3] J. Mirkovic and P. Peterson, "Class Capture-the-Flag Exercises," in *USENIX 3GSE*, August 2014.

[4] "PicoCTF," `https://www.picoctf.com`.

[5] "HSCTF," `http://www.hsctf.com`.

[6] P. Chapman, J. Burket, and D. Brumley, "PicoCTF: A Game-Based Computer Security Competition for High School Students," in *USENIX 3GSE*, August 2014.

[7] M. Olano, A. Sherman, L. Oliva, R. Cox, D. Firestone, O. Kubik, M. Patil, J. Seymour, I. Sohn, and D. Thomas, "SecurityEmpire: Development and Evaluation of a Digital Game to Promote Cybersecurity Education," in *USENIX 3GSE*, August 2014.

[8] "The Matasano Crypto Challenges," `http://cryptopals.com/`.

[9] "OverTheWire Wargames," `http://overthewire.org/`.

[10] "Embedded Security CTF," `http://microcorruption.com/`.

[11] W. Feng, "CS 492/592: Malware," `http://thefengs.com/wuchang/courses/cs492`.

[12] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, No Starch Press, 2012.

[13] P. Kirschner, J. Sweller, and R. Clark, "Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-based, Experiential, and Inquiry-based Teaching," *Educational Psychologist*, vol. 2, no. 41, pp. 75–86, 2006.

[14] S. Englemann, "Relating Operant Techniques to Programming and Teaching," *Journal of School Psychology*, , no. 6, 1968.

[15] J. Tuovinen and J. Sweller, "A Comparison of Cognitive Load Associated with Discovery Learning and Worked Examples," *Journal of Educational Psychology*, vol. 91, no. 2, 1999.

[16] T. Ptacek, H. Nielsen, and N. Carlini, "microcorruption: Security games to qualify and hire job candidates," in *USENIX 3GSE*, August 2014.

[17] N. Harbour, "Advanced Software Armoring and Polymorphic Kung-Fu," in *DEF CON*, August 2008.

[18] Oreans Technologies, "Themida," `http://www.oreans.com/`.

[19] UPX, "UPX: the Ultimate Packer for eXecutables," `http://upx.sourceforge.net`.