This set includes a number of games that can be incorporated into lectures or used as study aids. The purpose of these card games is to help students see the mechanics of various security topics in a simplified model, without the baggage of a full programming framework and memory model.

Because of the broad nature of the subject matter, we divide the topics into multiple largely disjoint card sets. Some of these topics are focused on binary data, and in particular on a toy computing architecture; these will share a single set of cards. Another set focuses on a networked application framework, which comprises a second set. A third set focuses on the network infrastructure.

We divide the games into a number of sections, dealing with specific topics. Not all courses will cover all of the sections included here, and doubtless there will be topics not included in these games. The section order here corresponds to the order of topics covered in the University of Maryland course CMSC 414: Computer and Network Security.

Each game should take 5 to 10 minutes in a classroom setting. Where appropriate, we provide variations that might take longer for use in studying. The games are designed for 4 to 6 players.

In addition to the contents of the game set, you might find paper and writing instruments helpful for some of the games.

1 Low-Level Programming Errors

These games cover topics like stack layout, buffer overflows, and some mitigation techniques. We will work in a pseudoassembly environment for a very simple architecture. Specifically, our architecture will have 4-bit bytes, which we will refer to as quartets (in comparison with 8-bit octets).

Our architecture has a very limited instruction set, and only three types of values:

- 1-quartet register IDs (1–7)
- 2-quartet big-endian literals (0x00–0xFF)
- 4-quartet addresses (0x0000–0x7FFF)
Here is a complete table of quartets and how they might be interpreted, based on context:

<table>
<thead>
<tr>
<th>quartet</th>
<th>value type</th>
<th>register</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>esp</td>
<td>add</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>ebp</td>
<td>sub</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>eip</td>
<td>xor</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>eax</td>
<td>jmp</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>ebx</td>
<td>jgz</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>ecx</td>
<td>jne</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>edx</td>
<td>ret</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>eex</td>
<td>call</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>efx</td>
<td>nop</td>
</tr>
<tr>
<td>A</td>
<td>-</td>
<td>-</td>
<td>load</td>
</tr>
<tr>
<td>B</td>
<td>-</td>
<td>-</td>
<td>store</td>
</tr>
<tr>
<td>C</td>
<td>register</td>
<td>-</td>
<td>push</td>
</tr>
<tr>
<td>D</td>
<td>literal</td>
<td>-</td>
<td>pop</td>
</tr>
<tr>
<td>E</td>
<td>address</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>F</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The add, sub, and xor opcodes take a register and an rvalue as the next two items. The rvalue is specified with a type, followed by the octets to specify the actual value. To add eax and ebx, storing the result in eax, you would have the following sequence of quartets: 14C5

Let’s break this down:

1. This is an add
2. The opcode takes a register as its first argument, which in this case is eax
3. The opcode takes an rvalue as its second argument, so this specifies that the value type is another register
4. The register to use for the second argument is ebx

If instead we want to add 1 to eax, we would instead have: 14D01

1. add
2. eax
3. D We now have a literal as the second argument, which is a two-quartet big-endian number
4. 01 This corresponds to the value $16 \times 0 + 1 = 1$

Here’s another example: 5E01C4C6

5. We’re going to jump to an address if the given value is greater than 0
6. The first argument is a register or address, and this specifies that it is an address
7. 01C4 The address is 0x01C4
8. C The second argument is an rvalue, which in this case is a register
9. 0 This is register ecx

That is, if register ecx stores a value greater than 0, we jump to 0x01C4.

The following would generate an Illegal Instruction: 14CA. This is because the second argument to add is supposed to be a register (C), but the octet A does not correspond to any register in the system, so the machine can’t process this. Similarly, hitting a non-opcode quartet where an opcode is required will generate an Illegal Instruction, as will anything other than C, D, or E where an rvalue is expected. Trying to access a memory address greater than 0x7FFF will generate a Segmentation Fault.
1.1 Stack Organization

We’re going to start by constructing a simple stack. This stack will have one array variable, the saved ESP, and the return address. To make things easy, let’s say the highest address of our stack is 0x7FFF, and we have 8 quartets “above” us on the stack.

Further, let’s say our current return address is 0x0123, which puts our code somewhere in the heap. The variable will be an array that holds two literal values, each of which is 2 quartets; our architecture can initialize this to 0 for the moment.

Your stack should look something like this (you can do the layout differently if you like):

The cards with a “?” can be drawn randomly, if you like, since they don’t matter.
1.2 Writing to a Buffer

Now we're going to write to our 2-element array `a` from the previous stack. The cards you need are, in this order: 2, A, 2, A, which from here on we’ll shorten to 2A2A. When we write these into `a`, we get the following stack:

```
? ? ? ?
? ? ? ?
? ? ? ?
```

What is the new value of `a`?
1.3 Overflowing a Buffer

Our array $a$ holds two elements, but what if we were to write more? We're now going to prepare a longer array to copy into $a$, starting with the cards used in the previous game: `2A2A2A7FF899934C4`. Copy those into $a$, which should now give you the following stack:

```
    3  4  C  4
    9  9  9  9
    7  F  F  8
    2  A  2  A
```

What happens when the current function returns?
1.4 Preparing Shellcode

Buffer overflow attacks often rely on improperly protected string copying routines. Since `strcpy()` uses a null byte to determine the end of the input string, our shellcode cannot include any nulls (quartet 0). Sometimes we have a null in our code, often as part of an address or literal value, which means we have to remove this null in order to construct useable shellcode.

Consider the following snippet of code, in our architecture:

```
add %eax #12
sub %ebx #1
jne @7FE9 %eax %ebx
```

This will add 12 to the register eax, subtract 1 from register ebx, and then jump to a memory location in the stack if they are not equal.

Converting this to quartets, this is:

```
1 4 D0C
2 5 D01
6 E7FE9 C4 C5
```

Note that this has two nulls in it, in the literal arguments to add and sub.

Rewrite this shellcode to remove the nulls. There is more than one way to do this, so discuss approaches with your group.
1.5 Metamorphic Executable Code

The previous game focused on simple transformations of assembled instructions to remove nulls while preserving functionality. In this game we extend this to the techniques used in metamorphic viruses (see Hunting for Metamorphic, by Ször and Ferrie). As before, we want to preserve the high-level logic of the program, but now we want to substantially change what the program looks like.

Because we’re going to have more complex logic, we need to specify the addresses we’re using. Nulls are not a problem in this game. Our program is loaded into memory starting from address 0x0100. The top of our stack (that is, %esp) is 0x7FA0.

This is the corresponding C-like pseudocode:

```c
void foo(int a, int b) {
    if ( a < 10 ) {
        syscall_1(a,b);
    }
}
```

There are no local variables, so we have the following stack elements:

<table>
<thead>
<tr>
<th>Element</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0x7FAA</td>
<td>5 (0x05)</td>
</tr>
<tr>
<td>b</td>
<td>0x7FA8</td>
<td>20 (0x14)</td>
</tr>
<tr>
<td>return</td>
<td>0x7FA4</td>
<td>0x00F0</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x7FA0</td>
<td>0x7FAD</td>
</tr>
</tbody>
</table>

When compiled, and then assembled, we have:

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Effect</th>
<th>Address</th>
<th>Quartets</th>
</tr>
</thead>
<tbody>
<tr>
<td>load %eex %esp</td>
<td>load value of %esp into %eex</td>
<td>0x0100</td>
<td>A 8 C1</td>
</tr>
<tr>
<td>add %eex #10</td>
<td>calculate address of a</td>
<td>0x0104</td>
<td>1 8 D0A</td>
</tr>
<tr>
<td>load %eax %eex</td>
<td>load value of a into %eax</td>
<td>0x0109</td>
<td>A 4 C8</td>
</tr>
<tr>
<td>push %eax</td>
<td>push value of a as first argument for call</td>
<td>0x010D</td>
<td>C 4</td>
</tr>
<tr>
<td>sub %eex #2</td>
<td>calculate address of b</td>
<td>0x010F</td>
<td>2 8 D02</td>
</tr>
<tr>
<td>load %ebx %eex</td>
<td>load value of b into %ebx</td>
<td>0x0114</td>
<td>A 5 C8</td>
</tr>
<tr>
<td>sub %eax #9</td>
<td>subtract 9 from a</td>
<td>0x0118</td>
<td>2 4 D09</td>
</tr>
<tr>
<td>load %efx %eip</td>
<td>load current instruction pointer into %efx</td>
<td>0x011D</td>
<td>A 9 C3</td>
</tr>
<tr>
<td>add %efx #22</td>
<td>calculate end of if block</td>
<td>0x0121</td>
<td>1 9 D16</td>
</tr>
<tr>
<td>jgz %efx %eax</td>
<td>jump past block if a-9&gt;0</td>
<td>0x0126</td>
<td>5 C9 C4</td>
</tr>
<tr>
<td>push %ebx</td>
<td>push value of b as second argument for call</td>
<td>0x012B</td>
<td>C 5</td>
</tr>
<tr>
<td>call $0x0010</td>
<td>call syscall_1 at address 0x0010</td>
<td>0x012D</td>
<td>8 E0010</td>
</tr>
<tr>
<td>ret</td>
<td>end of subroutine</td>
<td>0x0133</td>
<td>7</td>
</tr>
</tbody>
</table>

Try to come up with at least 3 metamorphic variations of this program, so that it has the same result but will foil a direct binary signature checker.
2 Web-Based Attacks

2.1 SQL Injection

For this, we will use the Web Attacks cards, W0–W16. Start with the Server (W0) and Attacker (W2) roles. The Server has an interface with an Input Field (W4).

The Attacker can use any of the Input, Attack, and Attack Effect cards (W6–W11). Try different Attack Effects, and discuss how the Server would respond after using the Process Form (W12) card.

2.2 SQLi Prevention

This builds off the previous game. The Server is now going to use the Server cards (W12–W16) to prevent the SQL Injection attack. Discuss how these additional Server actions protect against SQLi.

2.3 Stored XSS

Implement the following attack scenario in your group:

Client \[\text{Attacker} \rightarrow \text{Server}\]

\[
\text{coolguy.avatar}=\text{\textless img src=...}...\\
\text{from=coolguy, comment=Cool article!}\\
\text{load comments}\\
\text{Comment from coolguy <img src=\textless script>...}
\]

How might you defend against this, both at the Server and the Client?
2.4 Reflected XSS

Implement the following attack scenario in your group:

Client  Attacker  Server

http://victim.com/search.php?term=<script>...

Results for term "<script>..."

malicious request

How might you defend against this, both at the Server and the Client?

2.5 Cross-Site Request Forgery

Implement the following attack scenario in your group:

Client  Attacker  Server

<img src="http://bank.com/transfer.cgi?amt=9999&to=attacker">

acct += 9999
acct -= 9999

How might you defend against this, both at the Server and the Client?
2.6 Man-in-the-Middle

2.7 Remote File Inclusion

2.8 Defending Against XSS and CSRF
3 Cryptography

These games will use the Binary cards.

3.1 Random Oracle

In addition to the Quartet cards, you will also need B20 (Hash).

The Random Oracle produces a random output the first time it sees an input, and the same output on subsequent times. You can simulate this by creating a message either by selecting or randomly drawing cards from the deck of quartets. Now shuffle one of each of the quartets together as your oracle deck.

The hash function $H$ takes two quartets as input, and outputs a single quartet. To hash a longer message to a single quartet, you can begin with a 0 as the first input, and the first quartet of the message as the second. If you have not seen this pair of inputs before, draw a random card from the oracle deck and record this input and output. As you progress through the message, take the previous hash output as the first input.

There will be a lot of collisions with a 4-bit output; a decent hash function would not have as many.

3.2 Feistel Ciphers

We will use a Random Oracle and the Luby-Rackoff Result to construct a Feistel cipher. This requires all of the XOR (B19) and Hash (B20) cards. Instead of a progressive hash of a long message, our Random Oracle will operate on a fixed input for $\alpha$, and a single quartet at a time. In other words, our Feistel block size is 2 quartets, broken into left and right quartets.

Set up a Feistel cipher network with four blocks. The key quartets $K_1$ through $K_4$ should be selected beforehand, and will serve as the first argument to the hash function.

![Diagram](image)

Encryption block:

After the 4th block, swap the left and right halves.

Verify that after encrypting a message, you can successfully decrypt it.
3.3 S-P Networks

4 Network Security