Security

Dr. Michael Marsh
Department of Computer Science
University of Maryland

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>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A</td>
<td>-</td>
<td>ebp</td>
<td>sub</td>
</tr>
<tr>
<td>B</td>
<td>-</td>
<td>eip</td>
<td>xor</td>
</tr>
<tr>
<td>C</td>
<td>register</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>D</td>
<td>literal</td>
<td>-</td>
<td>-</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 rval 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. 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: 501C4C6

5. We're going to jump to an address if the given value is greater than 0
6. 01C4 The first argument is an address, in this case 0x01C4
7. C The second argument is an rval, which in this case is a register
8. 01 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*: 14C8. This is because the second argument to add is supposed to be a register (C), but the octet 8 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 rval is expected. Trying to access a memory address greater than 0xFF will generate a *Segmentation Fault*. 2
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 top 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):

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>F</td>
<td>F</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

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:

```
| ? | ? | ? | ? |
| ? | ? | ? | ? |
| ret | 0 | 1 | 2 | 3 |
| 7 | F | F | 8 |
```

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: 2A2A2A7FF8999934C4. 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
    2  A  2  A
```

What happens when the current function returns?
1.4 Preparing Shellcode

1.5 Executing Shellcode

1.6 Metamorphic Executable Code

I'm putting this here, because it's not enough for its own section, and more or less fits thematically.

2 Web-Based Attacks

multiple players is more relevant here, since we can split up client, server, attacker, and victim — not every game will have all of these, and this is still less than a full table's worth

the cards will be somewhat generic, so some of these will look very similar

need generic terminator for all injections

need supplied-field

need document/interface framing cards

need server response cards (like search, echo, etc.)

do we need data flow cards? (eg, client -> server, attacker -> victim, ...)

need attack effects: insertion, extraction, modification; composite with other actions (redirect, etc)

2.1 SQL Injection

2.2 SQLi Prevention

this will just have one or two extra cards for players to use
2.3 Man-in-the-Middle

2.4 Remote File Inclusion

2.5 Stored Cross-Site Scripting

2.6 Reflected Cross-Site Scripting

2.7 Cross-Site Request Forgery

2.8 Defending Against XSS and CSRF

this will just have one or two extra cards for players to use