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:

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

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`
- `4` The opcode takes a register as its first argument, which in this case is `eax`
- `C` The opcode takes an rval as its second argument, so this specifies that the value type is another register
- `5` 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`
- `4` `eax`
- `D` We now have a literal as the second argument, which is a two-quartet big-endian number
- `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
- `E` The first argument is a register or address, and this specifies that it is an address
- `01C4` The address is `0x01C4`
- `C` The second argument is an rval, which in this case is a register
- `6` This is register `ecx`

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

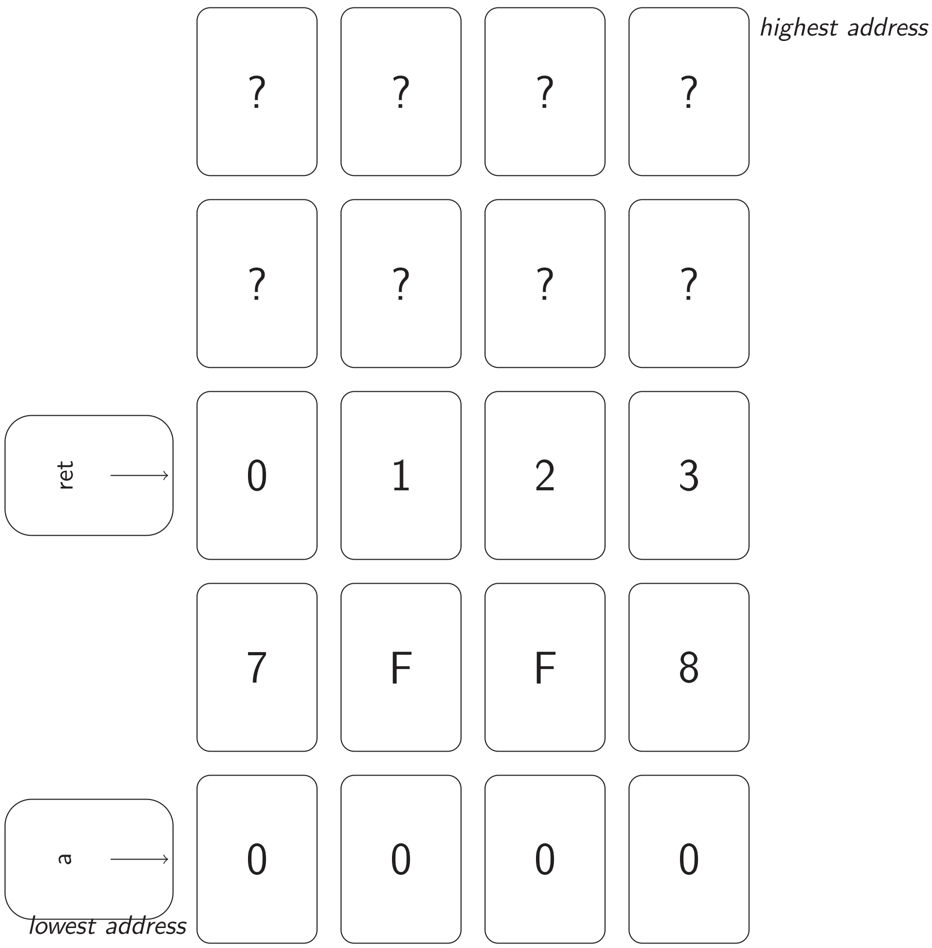
The following would generate an *Illegal Instruction*: `14CA`. This is because the second argument to `add` is supposed to be a register (`C`), but the octet `A` does not correspond to any register in the system, so the machine can't process this. Similarly, hitting a non-opcode quartet where an opcode is required will generate an Illegal Instruction, as will anything other than `C`, `D`, or `E` where an rval 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.

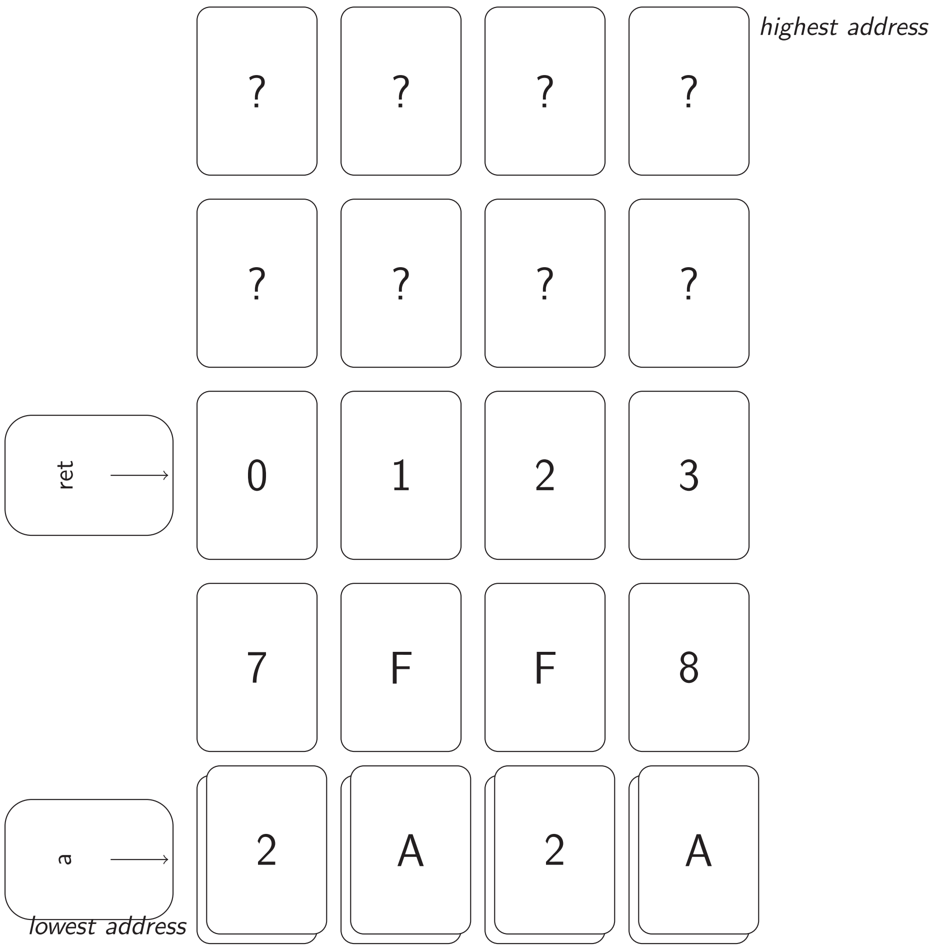
Your stack should look something like this (you can do the layout differently if you like):



The cards with a "?" can be drawn randomly, if you like, since they don't matter.

1.2 Writing to a Buffer

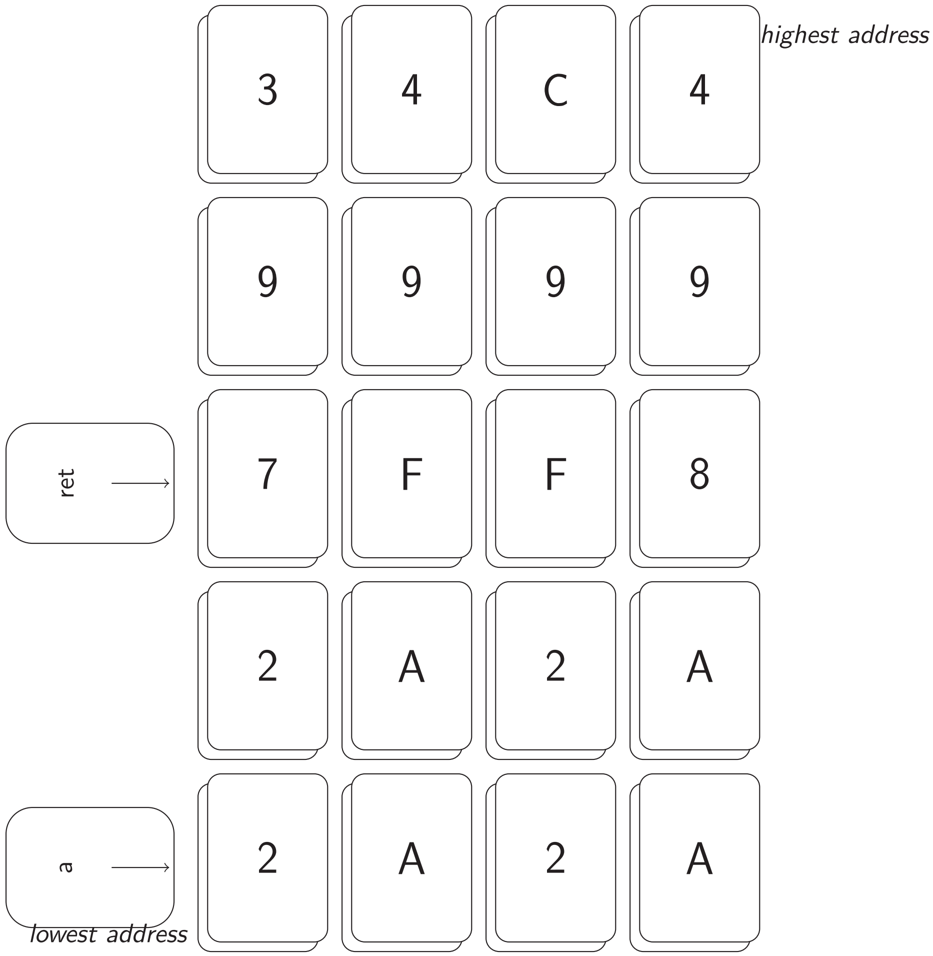
Now we're going to write to our 2-element array a from the previous stack. The cards you need are, in this order: 2, A, 2, A, which from here on we'll shorten to 2A2A. When we write these into a, we get the following stack:



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: 2A2A2A2A7FF8999934C4. Copy those into a, which should now give you the following stack:



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:

```
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:

Element	Address	Value
a	0x7FAA	5 (0x05)
b	0x7FA8	20 (0x14)
return	0x7FA4	0x00F0
%ebp	0x7FA0	0x7FAD

When compiled, and then assembled, we have:

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

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).

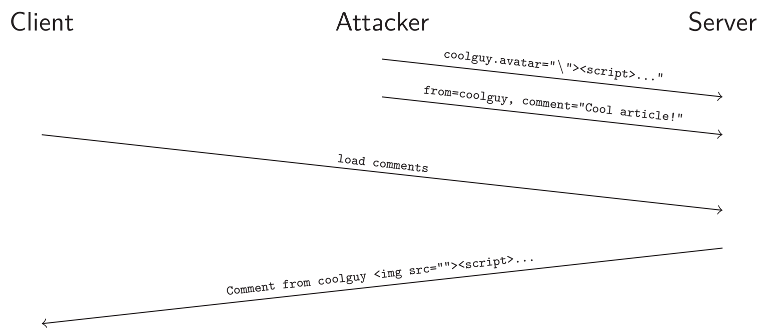
The Attacker can use any of the Input, Attack, and Attack Effect cards (W6–W11). Try different Attack Effects, and discuss how the Server would respond after using the Process Form (W12) card.

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

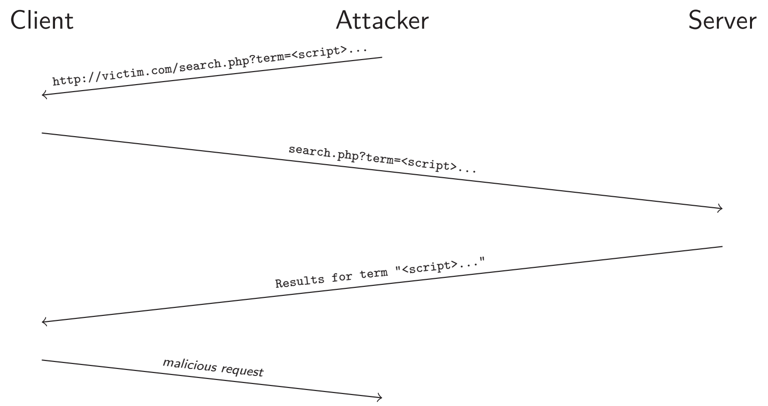
Implement the following attack scenario in your group:



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

2.4 Reflected XSS

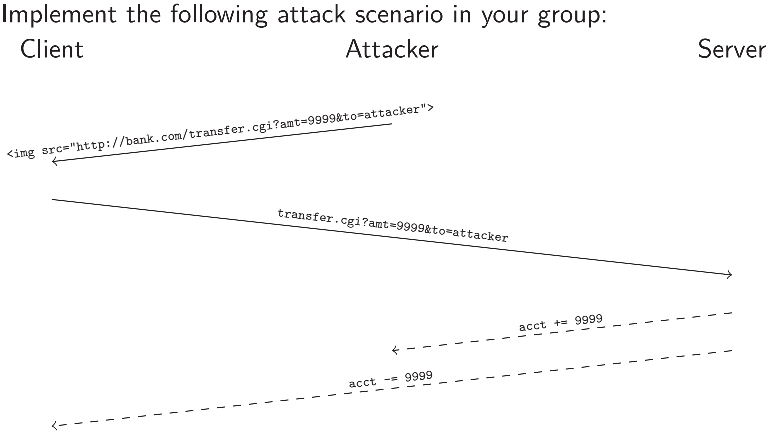
Implement the following attack scenario in your group:



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:



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.

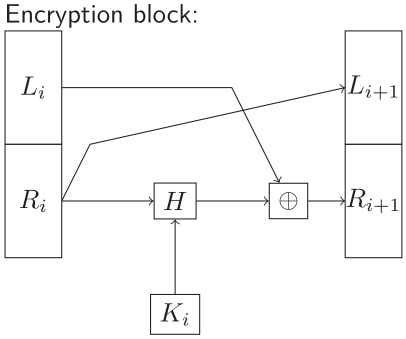
There will be a lot of collisions with a 4-bit output; a decent hash function would not have as many.

3.2 Fesitel Ciphers

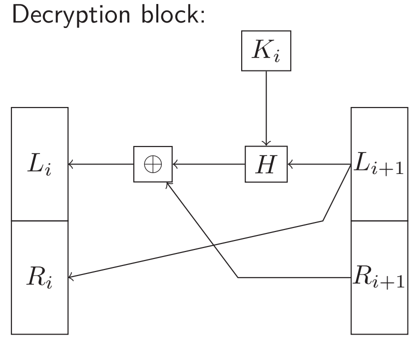
We will use a Random Oracle and the *Luby-Rackoff Result* to construct a Feistel cipher. This requires all of the *XOR* (B19) and *Hash* (B20) cards. Instead of a progressive hash of a long message, our Random Oracle will operate on a fixed input for a, and a single quartet at a time. In other words, 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.

Encryption block:



Decryption 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