# CS 380S - 0x1A Great Papers in Computer Security
# Fall 2012

## Homework #1

Due: 2pm CDT (in class), Sep 27, 2012

**NO LATE SUBMISSIONS WILL BE ACCEPTED**

**YOUR NAME:** _____

## Collaboration policy

**No collaboration** is permitted on this assignment. Any cheating (*e.g.*, submitting another person's work as your own, or permitting your work to be copied) will automatically result in a failing grade.

# Homework #1 (30 points)

## Problem 1

Molvanîan Security Defense Operating System (MS-DOS) protects against memory corruption exploits as follows. MS-DOS ensures that heap, stack, and code sections of the program are all page-aligned. On a 32-bit x86 machine (the only kind they use in Molvanîa), each page is 4096 bytes. MS-DOS randomizes memory layout using a modified runtime loader: each time a binary is loaded in the process address space, the loader selects a random page-aligned address for the code, data, and heap segments.

## Problem 1a (1 point)

What is this defense based on?

## Problem 1b (2 points)

The famous Molvanîan hacker R00tkowski has found an exploitable stack-based buffer overflow in the MS-DOS login program and decides to exploit it using a `return-to-libc` attack. He must correctly guess the buffer start address, as well as the location of `execve` call (which is the target of the control transfer in his attack) in the system call table. If the guess is incorrect, the login program crashes and then automatically restarts.

How many guesses are necessary, on average, to ensure a successful attack? Explain.

## Problem 2

All Molvanîan C compilers for x86 insert stack canaries into generated code to prevent stack-smashing attacks. Nevertheless, Molvanîan Cyber-Security Bureau mandates the use of `libsafe` with all executables compiled from C.

`libsafe` is a wrapper around the C string library, intended to ensure that string operations cannot overwrite any control information stored on the stack (such as saved return address, saved frame pointer, etc.). For example, the `libsafe` wrapper around `strcpy` adds the following check before `strcpy(src,dest)` is executed:

```
|framePointer - dest| > strlen(src)
```

**Problem 2a (3 points)**

What additional protections are gained by using `libsafe` with canary-equipped executables?

**Problem 2b (2 points)**

Give a short snippet of C code that contains a single call to a `libsafe`-protected `strcpy`, and yet is vulnerable to a memory corruption attack as a result of this call. Your attack must also bypass compiler-inserted stack canaries.

## Problem 3

x68 is Molvanîan homegrown chip architecture. Unlike on x86, the stack on x68 grows upwards.

### Problem 3a (2 points)

How does a stack-based overflow attack work on x68?

### Problem 3b (2 points)

How would you implement StackGuard on x68? What would be the main differences from x86?

### Problem 3c (2 points)

How would you implement `libsafe` on x68? What would be the main differences from x86?

## Problem 4 (4 points)

Sandboxing x86 (as opposed to RISC) code is difficult because variable-length x86 instructions are hard for the verifier to parse. In Molvanîa, however, code is shipped as ASCII assembly language source. The verifier modifies the source to ensure the following properties:

- Register %edx is only used to define the logical fault domain (*i.e.*, for segment matching).
- Every instruction that modifies memory...
  - ...is relative to the stack or frame pointer (with a small enough offset not to go out of bounds), or
  - ...uses the %edx register for segment matching with the following code:

    ```
    mov DEST, %edx
    bound %edx, domainrange
    INST SRC, (%edx)
    ```

    Here DEST is the memory location that is being written, domainrange is the location of pointers to the beginning and end of the fault domain's data segment, the `bound` instruction traps if %edx is not within that range, and INST and SRC can be any instruction and source that modify the memory pointed to by DEST.

The verifier allows relative branches to any byte within the fault domain's code segment, but control transfers outside the code segment use a jump table which transfers to trusted code stubs not in the code segment.

Explain how, even with this scheme, malicious code could escape the sandbox and modify other regions of a process's address space.

## Problem 5

### Problem 5a (3 points)

What is the exact control-flow property that Native Client enforces? Is it weaker, stronger, or the same as control-flow integrity?

### Problem 5b (2 points)

In Native Client, "springboards" are snippets of trusted code which are located in the memory of the untrusted binary module. Their purpose is to enable control transfers from the trusted runtime environment to untrusted code. Because the sprinboard code is trusted, it may include privileged instructions which are not normally available to the untrusted code.

What prevents untrusted code from executing these instructions by passing control—via either a jump, or sequential execution—to the springboard code located in its memory?

## Problem 6

Consider the following snippet of C code:

```
static char logfile[]="log.txt";

int checklog() {
  return (access(logfile, O_RDWR));
}
int openlog() {
  return (open(logfile, O_RDWR));
}
int writelog(int fd, int len) {
  if (fd > 0)
     write(fd, "", len);
}
void log() {
```

```
  int fd;
  if (checklog()) {
     fd=openlog();
     for(int i=0; i<5; i++)
        writelog(fd,64);
     writelog(fd,32);
  }
}

}
```

**Problem 6a (6 points)**

Write the Dyck model of the control flow of this code. If you are unable to write the Dyck model, write the callgraph and explain why it is imprecise and why a Dyck model would provide more precision.

**Problem 6b (1 points)**

Can you use a reference monitor based on the above Dyck model to prevent TOCTTOU attacks on this code? If yes, explain how; if no, explain why.