CS 380S - Theory and Practice of Secure Systems
Fall 2009

Homework #1

Due: 2pm CDT (in class), October 1, 2009

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. The Computer Sciences department code of conduct can be found at `http://www.cs.utexas.edu/users/ear/CodeOfConduct.html`

Homework #1 (30 points)

Problem 1

A famous Norwegian hacker OpKødë proposes the following defense against buffer overflow exploits on big-endian architectures. All code pages, including system library routines, should be mapped to low memory addresses (*e.g.*, from `00000000` to `000FFFFF`), and the rest of the pages should be marked as non-executable.

Problem 1a (2 points)

What is the principle behind this defense?

Problem 1b (2 points)

Would this technique prevent buffer overflow exploits? How about `return-to-libc` exploits? Explain.

Problem 1c (1 point)

What are the advantages and disadvantages of protecting memory in this way?

Problem 2 (3 points)

Suppose the system is protected by $W \oplus X$ (DEP). All C code is compiled with random canaries on the stack and safe exception handling is deployed. There are no applications (such as the Java Virtual Machine) that need readable-writeable-executable memory.

Write a fragment of C code that is vulnerable to a memory exploit and explain in detail how it enables a clever attacker to take complete control over the system.

Problem 3 (2 points)

Suppose the Flash plugin implementing the ActionScript Virtual Machine had been recompiled using CCured. Would this have prevented the ActionScript exploit? Explain.

Problem 4

Consider the following snippet of C code. Your goal is to manually analyze it using the CSSV technique.

```
#define  MAXARGS     10
#define  MAXALLOC   128
#define  MAXARGLEN  256

typedef struct arglist {
   char *arg[MAXARGS];
} arglist;

void main() {
  char *buf;

  buf = (char *)malloc(MAXALLOC);

  return fillbuf(buf);
}

int getarg(arglist **al, short index) {
  char *inputbuf;

  inputbuf = (char *)malloc(MAXARGLEN);

  fgets(inputbuf, MAXARGLEN, stdin);

  (*al)->arg[index] = inputbuf;
```

```
  return strcasecmp("END", inputbuf) ? 0 : 1;
}

int fillbuf(char *target) {
 short i = 0;
 arglist *aList = NULL;

 if (target) *target = '\0';

 while (getarg(&aList, i) && (i < MAXARGS)) {

    if (target != NULL)

       strcpy(target, aList->arg[i++]);

    ...
 }
}
```

Problem 4a (4 points)

Draw the abstract `points-to` state for this program.

Problem 4b (5 points)

Write the integer program corresponding to this code fragment, including all memory safety assertions.

Problem 4c (2 points)

Explain how integer analysis will detect a potential buffer overflow in this code.

Problem 5 (2 points)

Symbolic links in Unix are notoriously vulnerable to TOCTTOU attacks. To avoid problems with symbolic links, suppose that we implement the following pseudocode for removing a file:

```
lstat(filename, fileinfo);
if (fileinfo is not a symbolic link)
   rm filename;
```

Do you see any problems with this solution?

Problem 6

Compare and contrast two techniques for preventing memory error exploits: XFI (as described in "XFI: Software Guards for System Address Spaces," OSDI 2006) and WIT (as described in "Preventing Memory Error Exploits with WIT," IEEE S&P 2008).

Problem 6a (2 points)

Explain at least two advantages of the XFI approach over the WIT approach. If you believe that XFI has no advantages over WIT, make a technical case for this.

Problem 6b (2 points)

Explain at least two advantages of the WIT approach over the XFI approach. If you believe that WIT has no advantages over XFI, make a technical case for this.

Problem 6c (1 point)

Describe a realistic attack that may enable an attacker to compromise an XFI- or WIT-protected application or cause it to behave in a way not intended by the application's developer. Explain how the attack works and why it succeeds.

Problem 7 (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?