CS 6431 - Security and Privacy Technologies
Fall 2014

Homework #1

Due: 7:30pm EDT, October 8, 2014

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 (50 points)

Molvanîa is a small, land-locked republic in Eastern Europe famous for its phishers, spamlords, botmasters—and computer security researchers. It also produces 83% of the world's b33tr00t. Most people get to Molvanîa either by air or by accident, but in this homework, we travel there virtually.

Problem 1

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 1a (4 points)

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

Problem 1b (4 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 2

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

Problem 2a (3 points)

How does a stack overflow attack work on x68?

Problem 2b (3 points)

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

Problem 2c (3 points)

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

Problem 3 (6 points)

Consider the following snippet of C code:

```
1  char *p1 = malloc(90);
2  char *p2 = p1 + A;
3  char *p3 = p2 + B;
4  *p3 = '\0';
```

Suppose this code is protected from buffer overflows by "baggy bounds" checking with 16-byte slots. For each of the following values of A and B, indicate whether the checks will abort the program and, if so, at which line of the above code.

- A = 95, B = -10
- A = -10, B = 95
- A = 130, B = -5
- A = -5, B = 130
- A = 140, B = -15
- A = 135, B = -5

Problem 4 (6 points)

Molvanîan Cyber-Security Bureau requires that all C code be statically checked to verify that it satisfies a certain set of rules. Each rule is expressed by a finite-state automaton with a special ERROR state. As the checker scans the code, it keeps track of the current state in the automaton. If a state labelled ERROR is ever reached, then the checker reports an error in the code.

Draw finite-state automata representing the following security rules. If you believe the rule cannot be expressed by a finite-state automaton, explain why.

- Immediately before each call to `strcpy(dest,src)`, the program must check the length of `src` by calling `strlen(src)`.

- Each temporary file used by the program must be created using `mkstemp()`, written, and eventually closed.

- The return value of every call to `malloc` must be immediately checked to ensure that it is not NULL.

Problem 5

Workstations based on DEC Alpha architecture are still widely used in Molvanîa. In the DEC Alpha assembly language, all instructions are 4-bytes wide and must start on an aligned 4-byte boundary. Here are some examples:

- `br` *Ra*, *disp*
  An unconditional relative branch. This instruction stores the address of the next instruction in *Ra* and then skips *disp* instructions, where *disp* may be negative. For example, `br r13, -5` jumps back 5 instructions (this may happen in a loop, for example).

- `jmp` *Ra*, (*Rb*)
  Jump to register. Stores the address of the next instruction in *Ra*, then jumps and starts executing code at address *Rb*.

- `ldq` *Rv*, *disp* (*Ra*)
  Load. Takes the memory address contained in register *Ra*, adds *disp* to it, and loads the value of the memory location at this address into register *Rv*.

- `stq` *Rv*, *disp* (*Ra*)
  Store. Takes the memory address contained in register *Ra*, adds *disp* to it, and stores the value of register *Rv* into the memory location at this address.

- `bis` *Ra*, *Rb*, *Rc*
  Compute bitwise OR of *Ra* and *Rb* and store it into *Rc*.

- `and` *Ra*, *Rb*, *Rc*
  Compute bitwise AND of *Ra* and *Rb* and store it into *Rc*.

Problem 5a (3 points)

Fault isolation requires inserting special checking code before every *unsafe* instruction, *i.e.*, an instruction that may potentially write or execute memory outside the fault domain. For example, a store instruction `stq` *Ra*, 0(*Rb*) is unsafe if it cannot be statically checked that the address contained in *Rb* is within the fault domain's data segment.

In the following list, circle the instruction(s) which can be unsafe:

- `br` *Ra*, *disp* where *disp* falls within the fault domain's code segment.
- `jmp` *Ra*, (*Rb*)
- `ldq` *Rv*, *disp* (*Ra*)
- `bis` *Ra*, *Rb*, *Rc*

Problem 5b (3 points)

Suppose that the unsafe store instructions are "sandboxed" as follows. We use dedicated registers $r20$ and $r21$ to store, in the positions corresponding to the segment identifier part of a memory address, all-zero bits and the segment ID bits, respectively. If the code contains an unsafe store instruction `stq r2, 0(r1)`, it is replaced by the following three instructions:

```
and r1, r20, r1
bis r1, r21, r1
stq r2, 0(r1)
```

How can you subvert the safety of the system that uses this sandboxing mechanism?

Problem 5c (3 points)

Suppose communication between fault domains is implemented as follows. For each fault domain, the trusted execution environment inserts special "stubs" (little snippets of code) into a special region of that domain's code segment. Because the code of the stubs is trusted, it may contain unsafe instructions. Furthermore, the stubs are the only part of the fault

domain's code segment that is allowed to have instructions branching outside of this code segment.

When a trusted caller calls an untrusted function, it branches to the "entry" stub, which copies arguments, saves registers that must be changed when switching fault domains, and passes control to the untrusted code. When the untrusted code returns, it jumps directly to the "return" stub in its code segment, which restores the context and returns to the caller.

How can you subvert the safety of the system that uses this cross-domain communication mechanism?

Problem 5d (3 points)

How should you implement the "stubs" for cross-domain communication so that they cannot be subverted? You may explain or draw a picture.

Problem 6 (3 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 7

Problem 7a (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 7b (3 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?