# XFI

Required reading: XFI: software guards for system address spaces.

## Introduction

Problem: how to use untrusted code (an "extension") in a trusted program?

- Use untrusted jpeg codec in Web browser
- Use an untrusted driver in the kernel

What are the dangers?

- No fault isolations: extension modifies trusted code unintentionally
- No protection: extension causes a security hole
    - Extension has a buffer overrun problem
    - Extension calls trusted program's functions
    - Extensions calls a trusted program's functions that is allowed to call, but supplies "bad" arguments
    - Extensions calls privileged hardware instructions (when extending kernel)
    - Extensions reads data out of trusted program it shouldn't.

Possible solutions approaches:

- Run extension in its own address space with minimal privileges. Rely on hardware and operating system protection mechanism.
- Restrict the language in which the extension is written:
    - Packet filter language. Language is limited in its capabilities, and it easy to guarantee "safe" execution.
    - Type-safe language. Language runtime and compiler guarantee "safe" execution.
- Software-based sandboxing.

## Software-based sandboxing

Sandboxer. A compiler or binary-rewriter sandboxes all unsafe instructions in an extension by inserting additional instructions. For example, every indirect store is preceded by a few instructions that compute and check the target of the store at runtime.

Verifier. When the extension is loaded in the trusted program, the verifier checks if the extension is appropriately sandboxed (e.g., are all indirect stores sandboxed? does it call any privileged instructions?). If not, the extension is rejected. If yes, the extension is loaded, and can run. If the extension runs, the instruction that sandbox unsafe instructions check if the unsafe instruction is used in a safe way.

The verifier must be trusted, but the sandboxer doesn't. We can do without the verifier, if the trusted program can establish that the extension has been sandboxed by a trusted sandboxer.

The paper refers to this setup as instance of proof-carrying code.

# Software fault isolation

SFI by Wahbe et al. explored out to use sandboxing for fault isolation extensions; that is, use sandboxing to control that stores and jump stay within a specified memory range (i.e., they don't overwrite and jump into addresses in the trusted program unchecked). They implemented SFI for a RISC processor, which simplify things since memory can be written only by store instructions (other instructions modify registers). In addition, they assumed that there were plenty of registers, so that they can dedicate a few for sandboxing code.

The extension is loaded into a specific range (called a segment) within the trusted application's address space. The segment is identified by the upper bits of the addresses in the segment. Separate code and data segments are necessary to prevent an extension overwriting its code.

An unsafe instruction on the MIPS is an instruction that jumps or stores to an address that cannot be statically verified to be within the correct segment. Most control transfer operations, such program-counter relative can be statically verified. Stores to static variables often use an immediate addressing mode and can be statically verified. Indirect jumps and indirect stores are unsafe.

To sandbox those instructions the sandboxer could generate the following code for each unsafe instruction:

```
  DR0 <- target address
  R0 <- DR0 >> shift-register;  // load in R0 segment id of target
  CMP R0, segment-register;     // compare to segment id to segment's
ID
  BNE fault-isolation-error     // if not equal, branch to trusted
error code
  STORE using DR0
```

In this code, DR0, shift-register, and segment register are *dedicated*: they cannot be used by the extension code. The verifier must check if the extension doesn't use they registers. R0 is a scratch register, but doesn't have to be dedicated. The dedicated registers are necessary, because otherwise extension could load DR0 and jump to the STORE instruction directly, skipping the check.

This implementation costs 4 registers, and 4 additional instructions for each unsafe instruction. One could do better, however:

```
  DR0 <- target address & and-mask-register // mask segment ID from
target
  DR0 <- DR0 | segment register // insert this segment's ID
  STORE using DR0
```
This code just sets the write segment ID bits. It doesn't catch illegal addresses; it just ensures that illegal addresses are within the segment, harming the extension but no other code. Even if the extension jumps to the second instruction of this sandbox sequence, nothing bad will happen (because DR0 will already contain the correct segment ID).

Optimizations include:

- use guard zones for *store value, offset(reg)*
- treat SP as dedicated register (sandbox code that initializes it)
- etc.

# XFI

XFI extends SFI in several ways:

- Handles fault isolation and protection
- Uses control-folow integrity (CFI) to get good performance
- Doesn't use dedicated registers
- Use two stacks (a scoped stack and an allocation stack) and only allocation stack can be corrupted by buffer-overrun attacks. The scoped stack cannot via computed memory references.
- Uses a binary rewriter.
- Works for the x86

x86 is challenging, because limited registers and variable length of instructions. SFI technique won't work with x86 instruction set. For example if the binary contains:

```
  25 CD 80 00 00   # AND eax, 0x80CD
```
and an adversary can arrange to jump to the second byte, then the adversary calls system call on Linux, which has binary the binary representation CD 80. Thus, XFI must control execution flow.

XFI policy goals:

- Memory-access constraints (like SFI)
- Interface restrictions (extension has fixed entry and exit points)
- Scoped-stack integrity (calling stack is well formed)
- Simplified instructions semantics (remove dangerous instructions)
- System-environment integrity (ensure certain machine model invariants, such as x86 flags register cannot be modified)
- Control-flow integrity: execution must follow a static, expected control-flow graph. (enter at beginning of basic blocks)

- Program-data integrity (certain global variables in extension cannot be accessed via computed memory addresses)

The binary rewriter inserts guards to ensure these properties. The verifier check if the appropriate guards in place. The primary mechanisms used are:

- CFI guards on computed control-flow transfers (see figure 2)
- Two stacks
- Guards on computer memory accesses (see figure 3)
- Module header has a section that contain access permissions for region
- Binary rewriter, which performs intra-procedure analysis, and generates guards, code for stack use, and verification hints
- Verifier checks specific conditions per basic block. hints specify the verification state for the entry to each basic block, and at exit of basic block the verifier checks that the final state implies the verification state at entry to all possible successor basic blocks. (see figure 4)

Can XFI protect against the attack discussed in last lecture?

```
  unsigned int j;
  p=(unsigned char *)s->init_buf->data;
  j= *(p++);
  s->session->session_id_length=j;
  memcpy(s->session->session_id,p,j);
```

Where will *j* be located?

How about the following one from the paper *Beyond stack smashing: recent advances in exploiting buffer overruns*?

```
void f2b(void * arg, size_t len) {
  char buf[100];
  long val = ..;
  long *ptr = ..;
  extern void (*f)();

  memcopy(buff, arg, len);
  *ptr = val;
  f();
  ...
  return;
}
```

What code can *(*f)()* call? Code that the attacker inserted? Code in libc?

How about an attack that use *ptr* in the above code to overwrite a method's address in a class's dispatch table with an address of support function?

How about data-only attacks? For example, attacker overwrites *pw_uid* in the heap with 0 before the following code executes (when downloading /etc/passwd and then uploading it with a modified entry).

```
FILE *getdatasock( ... ) {
  seteuid(0);
  setsockeope ( ...);
  ...
  seteuid(pw->pw_uid);
  ...
}
```

How much does XFI slow down applications? How many more instructions are executed? (see Tables 1-4)