CMSC838z - Language-Based Security

2005-Feb-07 Discussion notes (blame Justin McCann) on:

- Efficient Software-Based Fault Isolation (Wahbe, et al)
- SASI Enforcement of Security Policies: A Retrospective (Erlingsson & Schneider)

Goal of “Efficient SFI” paper: limit faults in untrusted 3rd-party plugins/modules.

Traditional Solution: Hardware- and OS-based per-process separation and inter-process communication (IPC)– but context switches are too slow for modules needing frequent communication.

Proposed Solution: Use SFI to limit how code can write to memory and make jumps

Although unclear in the SFI paper, presumably the entire application and untrusted modules share a single process, which has multiple segments.

Untrusted code is limited by:

- Segment Matching and/or Address Sandboxing, which prevent writing and jumping across segments. SFI segment matching inserts checks before each jump, write, and store instruction to verify that the appropriate segment identifier is used, and aborts if a violation would occur. Sandboxing will allow instructions using illegal addresses to execute, but forces the addresses to remain within the appropriate segment.

- Forcing system calls to use cross-fault-domain RPC for arbitration of potential security violations. This essentially creates an OS-like execution monitor on top of SFI segment-based tamper proofing.

Since untrusted modules can only write to the data segment, and only execute from and jump to the code segment, the untrusted module cannot rewrite itself to fool the verifier and code monitor.

Both SFI and SASI make strong assumptions about the output of GCC and the way it limits the object code it creates, specifically in the way it creates jump labels. Without those patterns/limitations, both SFI and SASI have issues.

Five registers are set aside for use only by the SFI code:

- segment mask
- code segment identifier
- code address
- data segment identifier
- data address

Hicks’s proverbs/flamebait:

- “Type safety implies memory safety”
- The SFI approach doesn’t really bring stability (availability), since you still have a single process that dies whenever there is an errant write to memory. *(JNM– Could the read and write limits combined with forcing the untrusted code to work only in its legal segments prevent it from actually crashing? It could perform all sorts of bogus application-level errors, hopefully caught by the arbiter, but by definition wouldn’t be able to segfault.)*

- What happens if a third-party module grabs a lock that the application needs? *(JNM– Shouldn’t the arbitration code be written to prevent this? Seems like a job for SASI.)*

- The sandboxing techniques proposed in this paper led to the Java OS and HotJava web browser.

- The SFI approach doesn’t provide read protection like HW/OS based protection does– so the real performance hit \( (h) \) from the equation should be around 20% instead of 4.3%.

Registering OS timers to signal/interrupt the trusted application can prevent untrusted modules from running away with the processor, but doesn’t provide memory fault isolation.

SFI provides integrity naturally, and can provide confidentiality by adding read protection. Something more heavy-weight (with higher overhead) like per-thread or per-process separation is needed to provide availability.

**SFI performance calculation** indicates that software IPC provides performance gains over HW/OS based IPC/protection in almost all cases where IPC is needed.

**Levels of abstraction:**

- Both SASI and SFI seem to be limited by the ability to specify application-level security policies
- x86 SASI can’t abstract much above the assembly code level
- JVML SASI can abstract to level of function/method calls

Q: What would you need to do to implement SFI in Apache, for example? Hicks: Run automated tools to create arbitration code, to verify/rewrite third-party module code.

**Research Questions**

RQ: What does the curve look like when the encapsulation overhead \( (h) \) increases, e.g. by adding read protection?

RQ: How much of your tool set (compiler, the compiler that compiled the verifier, etc) must be a part of the TCB?

RQ: Could you construct a formal proof that SASI/SFI actually do what they claim to do?

RQ: Possibly fruitful research area: creating one-to-one mappings between application-level, syscall-level, and opcode-level security policies.

RQ: Has there been more work done on Security Automaton Language (SAL)?