Protection Mechanisms - A brief overview

ithilgore

sock-raw.org

16 Dec 2009

Index

1. IRM

What is a Reference Monitor?
A way to observe execution and contain/trap a possible violation of a previously defined security policy. Inline means it is placed inside the monitored untrusted process.

- Must not be circumvented.
- Must have low overhead.

1. IRM --[Example]

Canary - Buffer overflow on Stack protection

```
...      ...
 |------------------------------------|
 | parameters passed to function      |
 |------------------------------------|
 | function's return address (RET)    |
 |------------------------------------|
 | canary                             |
 |------------------------------------|
 | local frame pointer (%ebp)         |
 |------------------------------------|
 | local variables                    |
 |------------------------------------|
...      ...
```

2. Dynamic Information Flow Tracking (DIFT)

Use of hardware-implemented security tags to mark potentially malicious data as spurious (usually data coming from I/O channels that can be manipulated by an attacker) and raising an exception if they are used as an instruction or jump address.

- Protects against last step of attack -> executing attacker's code.
- Security Tag propagation mechanism -> [copy, comp, lda, sta] dependencies
- Efficient Tag Management -> multi-granularity — Page vector mask: [per page secure, per quadword, per byte, all spurious]

2. DIFT --[Example]

I/O from fgets is marked as spurious. Data written to by fgets (i.e buf) also marked as spurious. If saved return address is overwritten (by overflowing buf) it will be marked as spurious as well.

```
int func(char *fname)
{
    char buf[256];
    FILE *src;

    src = fopen(fname, "rt");
    while(fgets(buf, 1044, src)) {
        ...
    }
    return 0;
}
```

2. DIFT --[Pros/Cons]

Advantages:

1. Effective against most attacks:
   stack overflows, heap overflows, vudo/heap corruption, format string attacks
2. Low memory space and performance overhead

Disadvantages

1. Requires special processor and architecture for hardware tagging support.
2. No actual implementation yet
3. Legitimate cases of spurious data (jump tables, dyn func ptrs) -> requires additional binary inspection software algorithm to mark bound-checked such data as safe

3. Software Fault Isolation (SFI)

A method of software sandboxing mainly directed for use in modules-based applications. The process' modules reside in the same virtual space, yet are isolated from each other through software segmentation.

- use of segment register to define a fault domain (sandbox)
- check jump or store instructions with run-time resolved target addresses
- requires additional code to be prepended to each of these (binary rewriting)
- dedicated hw registers to protect from skipping checks
- communicate with trusted process through RPC (i.e shared mem)

3. SFI --[Example]

Before a function finishes (ret) (pop moves the saved eip return address into ebx)
---[Normal

```
pop %ebx
jmp *ebx
```

---[SFI applied

```
pop %ebx
mov %ebx, %ded_reg # use of dedicated register
cmp %ded_reg, seg_identifier # is addr inside fault domain?
jne trap # if not, trap
jmp *%ded_reg
```

3. SFI --[Pros/Cons]

Advantages:

1. Small overhead
2. Protects against some attacks

Disadvantages

1. Requires use of dedicated registers (not easy in architectures with small set of registers e.g i386)
2. Concept is lacking in specifying what happens with dynamically linked code (like libc). Is it considered to be inside the fault domain? If yes, ret2libc attacks will work.

3. SFI --[SASI]

Security Automata SFI Implementation
A generalization of SFI to define certain security policies, rather than just code flaw tampering protection. A formal language (security automata) is used to write policies in abstract form e.g app must not open more than 3 windows.

- It can't protect against advanced attacks without additional help.
- Use of partial evaluation using static analysis to eliminate unnecesarry checks (e.g "no division by 0" check only before DIV)

4. Advanced Sandboxing - seccomp

Seccomp is a feature of the Linux kernel that is enabled in (most) contemporary Linux distributions. It restricts a thread to a small number of system calls:

```
    * read()
    * write()
    * exit()
    * sigreturn()
```

If the thread calls any other system call, the entire process gets terminated. A trusted helper thread is invoked to inspect and run other allowed system calls on behalf of the sandboxed thread.

4. seccomp --[Syscall Interception]

Ideas

- Link against specially-built copy of glibc (too much maintenance cost)
- Find all places where glibc makes system calls and redirect to our wrapper function (preferred way)

Failure to rewrite all of them -> app termination
Syscall arguments are written to a socketpair() which are read by the trusted thread.

4. seccomp --[Mem access races 1/2]

TOCTTOU race condition
Syscall arguments deemed safe by trusted thread and just before syscall execution they are changed to malicious ones by the untrusted thread. Problem exists because of shared address space between threads.

Solution:
Use of extended registers (e.g SSE) for local variables of trusted thread (which should be coded in asm).

4. seccomp --[Mem access races 2/2]

For syscalls like fork(2) which need to pass a ptregs struct *ptr (residing in userspace) we need to invoke a separate trusted _process_

Trusted thread - process communicate through shared memory page.

A verified by the process data block (only accessible by trusted code) holds the syscall parameters which are read by the thread to finally execute the system call.

4. seccomp --[Considerations]

- A robust architecture is as important as bugless trusted code.
- Many attack vectors through race condition bugs and matters of trust.
- Module/plugin architecture is very common and thus a target for sandbox environments.
- Sandboxing will likely become a really sought-after research field.

5. Control Flow Integrity (CFI)

A general mitigation technique with a security policy which dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined ahead of time through source-code or binary analysis and/or execution profiling.

- Use of labels (special bit patterns) to mark possible allowed destinations for each control-flow transfer.
- Inserted checks ensure execution stays within CFG.

5. CFI --[Example 1/2]

Two destinations are equivalent when CFG contains edges to each from the same set of sources.

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

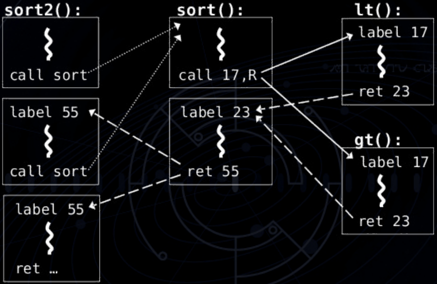
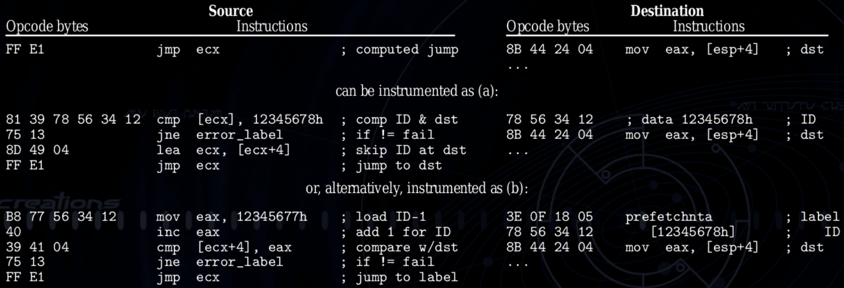


Figure 1: Example program fragment and an outline of its CFG and CFI instrumentation.

5. CFI --[Example 2/2]



```
                         Source                                              Destination
Opcode bytes               Instructions                         Opcode bytes        Instructions
FF E1                jmp  ecx                ; computed jump    8B 44 24 04    mov  eax, [esp+4]   ; dst
                                                                ...

                                        can be instrumented as (a):

81 39 78 56 34 12    cmp  [ecx], 12345678h   ; comp ID & dst    78 56 34 12    ; data 12345678h    ; ID
75 13                jne  error_label        ; if != fail       8B 44 24 04    mov  eax, [esp+4]   ; dst
8D 49 04             lea  ecx, [ecx+4]       ; skip ID at dst   ...
FF E1                jmp  ecx                ; jump to dst
                                   or, alternatively, instrumented as (b):
B8 77 56 34 12       mov  eax, 12345677h     ; load ID-1        3E 0F 18 05    prefetchnta         ; label
40                   inc  eax                ; add 1 for ID     78 56 34 12      [12345678h]       ;    ID
39 41 04             cmp  [ecx+4], eax       ; compare w/dst    8B 44 24 04    mov  eax, [esp+4]   ; dst
75 13                jne  error_label        ; if != fail       ...
FF E1                jmp  ecx                ; jump to label
```

Figure 2: Example CFI instrumentations of a source x86 instruction and one of its destinations.

3. CFI --[Pros/Cons]

Advantages:

1. Effective against most types of attacks
2. (finally sth that) protects against the ret2libc technique
3. Small runtime overhead

Disadvantages

1. Requires binary recompilation.
2. Preparation (static analysis) time overhead.
3. Doesn't protect against attacks that don't violate CFG (malicious arguments to syscalls, incorrect argument-string parsing etc)

References

1. MS Gleipnir
   http://research.microsoft.com/en-us/projects/Gleipnir/
2. Control-Flow Integrity: Principles, Implementations, and Applications
   http://research.microsoft.com/pubs/69217/ccs05-cfi.pdf
3. Efficient Software-Based Fault Isolation
   http://www.stanford.edu/class/cs295/papers/wahbe93efficient.pdf
4. SASI Enforcement of Security Policies: A Retrospective
   http://www.cs.cornell.edu/fbs/publications/sasiNSPW.ps
5. seccomp
   http://code.google.com/p/seccompsandbox/wiki/overview
6. Secure Program Execution via Dynamic Information Flow Tracking
   http://csg.csail.mit.edu/pubs/memos/Memo-467/memo-467.pdf
7. Bypassing Stackguard and Stackshield
   http://www.phrack.org/issues.html?issue=56&id=5

Questions?