September 14, 2011 – Inline Reference Monitors

Tom Magrino

1 A Static Analysis Example

We examine the following implementation of the factorial function and use static analysis to prove that the value returned is always a positive integer.

```
int fact(n) {
        int y = 1;
loop:
        if (n <= 0)
                goto done;
        y = y * n;
        n = n -1;
        goto loop;
done:
        return y;
}
```

We remove the function header, instead we add an additional line to the top of the code to nondeterministically assign a value to `n` (the argument of `fact`). This makes it so we're examining a simple series of instructions.

```
        /* Nondeterministically assign a value to n */
        int n = *;
        int y = 1;
loop:
        if (n <= 0)
                goto done;
        y = y * n;
        n = n -1;
        goto loop;
done:
        return y;
```

We continue by defining two boolean predicates:

pn – true if and only if the value of `n` is positive.

py – true if and only if the value of `y` is positive.

Now, we rewrite the code to use `pn` and `py` instead of `y` and `n`. For each operation involving `n` and `y`, we replace it by an expression that updates (and/or uses) the truth values of these predicates.

```
        bool pn = *;
        bool py = true;
loop:
        if (!pn)
                goto done;
        /* Can't always predict, this is a bit 'lossy.' */
        py = (py && pn) || (!py && !pn && *);
        /* Watch for the case of n == 1! */
        pn = (pn && *);
        goto loop;
done:
        /* We want to show that py must be true at this point */
```

Now we go through each line (following all branches in execution) keeping track of all of the possible values of `pn` and `py` after each line of code is executed (we continuously do this until we've reached a point where there are no more updates):

```
        bool pn = *;
        bool py = true;
        /* pn can be either true or false, py is true */
loop:   if (!pn) {
                /* pn is false, py is true */
                goto done;
        }
        /* pn is true, py is true */
        py = (py && pn) || (!py && !pn && *);
        /* pn is true, py is true */
        pn = (pn && *);
        /* pn can be either true or false, py is true */
        goto loop;
done:
        /* pn is false, py is true */
```

After getting to a point where there are no more updates (a fixed point process), we see that `py` is only true by the time factorial returns its result.

So the basic idea is to reduce to a finite state space (by reducing to boolean checks rather than a whole range of values) to simplify reasoning about the code. Because this removes info, this can cause false positives, but now that there's only finitely many values it's relatively simple to infer facts about the predicates.

2 A Brief History of SFI

In '93 there was an early paper about SFI for RISC archtectures to do sandboxing. However, it didn't work for the most popular architecture at the time, x86, because it was CISC. The reason they needed to use RISC was that fixed length instructions were a big part of the technique. For a long time this was it. It was considered to be a cute idea that didn't work for what we care about efficiently.

PittSFIeld Paper was the next major breakthrough. It worked by reducing to the RISC case by rewriting the code to be buffered. Huge breakthrough in performance but there was tons of variation in the running time and was still kind of expensive.

Later Google developed Native Client (NaCl) for use in a browser. It resulted in greater efficiency on x86-32. The idea was to combine the ideas present in PittSFIeld but using segment registers to create memory bounds without having the overhead of masking out addresses all the time (along with a few other optimizations). This resulted in more interest in these techniques from industry. The one downside was that it would only work for the 32 bit version of x86 (x86-64 does not have segment registers).

Later Google figured out how to do sandboxing in NaCl on the ARM and x86-64 architectures, making it a viable platform again.

3 SFI and Sandboxing Discussion

Up to now, we've been just mitigating software vulnerabilities. We have a new thread model now: running untrusted code. The question is "how do we do it safely?"

SFI is one of many approaches to Sandboxing. There's also

- Using process isolation.
- Using OS Support along with syscall filtering.
- Using privilege separation.

So we're imagining running a plugin written in native code for our browser (where they do this using a browser API and the browser calls the plugin to perform jobs). Single process browser for now, but we're pretty sure it still applies to multiprocess browsers.

Plugins will be really dangerous if we don't have a way of isolating the code. Most modern browsers use separate processes but as far as we know, none of them set different permissions on the separate process.

So why was NaCl so much better than previous work? We're not sure, there's a few factors at work here:

- They used different architectures.
- They used a implementation that was meant for production code unlike PittSFIeld which was a research prototype.

4 A Review of PittSFIeld

The security policies of PittSFIeld were:

- Untrusted code does not jump into the middle of trusted code or any data.
- Untrusted code should not be able to read and/or write trusted data.

All of this is done at user level through virtual memory.

How? We modify instructions. Here's an example of some transformations that would be done:

- `mov eax, (ebx)`

  will become

```
and 0x40FFFFFF, ebx
mov eax, (ebx)
```

- `jmp 0x103B2F20`

  Either will be left alone or removed since we can check this instruction statically.

- `jmp (eax)`

  will become

```
and 0x10FFFFE0, eax
jmp (eax)
```

By masking, it tries to force it to some address in the sandbox regions or a guard page, so that there are no accesses outside of the sandbox region. If the code is well behaved, this is not problematic. However, there are no promises beyond the security policy if the untrusted code does buggy accesses. This has better performance than doing a conditional test.

How do we sandbox more than one process at the same time? PittSFIeld does not handle this case and NaCl have some techniques which work, but it's not great.

So why did PittSFIeld use a 32 byte padding? They used the space for extra instructions to do sandboxing (like the mask instructions). This allows them to remove the worry of jumping just past the code that sandboxes a given instruction (so each 32 byte group is a "safe instruction" group).

So what happened with `ret` instructions? They didn't transform it to this:

```
and 0x10FFFFE0, (esp)
ret
```

Problem is context switch with another thread, which can modify esp between check and ret. So what it does instead is transform to something like this:

```
mov eax, (esp)
dec esp
and 0x10FFFFE0, eax
jmp (eax)
```

PittSFIeld exhibits a common security idea with the untrusted translator with trusted verifiers (instead of trusting the translator). The idea is that this reduces the amount of code that was trusted in the system.

```
    translator                  verifier
x86 -----------> sandboxed x86 ---------> verified and run (or abort)
    untrusted                   trusted
```

So they simplify verification by verifying the simpler code, instead of trying to control more of the system and leading to a larger trusted computing base.