CMPSCI 377 Operating Systems **Fall 2009**

Lecture 11

Lecturer: Emery Berger *Scribe: Tyler Robertson*

11.1 Worse Is Better

The essay is an observation on the sort of systems that gain mass adoption, and the sort of systems that don't. It describes two philosophies of system development:

MIT / Stanford Style

- implementations are simple and clean
- the interface is simple
- the system is elegent / has purity
- system correctness is required
- the system should be consistent
- the system should encompass the entire scope of the problem space

We note that this is a style favored by PL researchers and developers. These attributes combine to form very long development cycles, and oftentimes do not ever ship.

Examples of the MIT style include Scheme and Multix.

New Jersey / Berkeley Style

- simplicity of implementation is the prime directive
- the interface should be simple, but not at cost of the implementation
- consistency is nice, but can be sacrificed if it aids simplicity
- the system should encompass as many aspects of the problem space as is practical, but can be sacrificed for the sake of the above attributes

We note that this is a style favored by systems engineers and researchers. These attributes allow for smaller development cycles, thereby allowing for more rapid releases and iterations. Generally, these systems solve enough of a problem to be usable, and will get better as they are further developed.

Examples of the New Jersey style in Common Lisp, C, and Unix.

Tangent: the essay "The Cathedral and the Bazaar" was mentioned, and can be found here:

http://www.catb.org/esr/writings/cathedral-bazaar/cathedral-bazaar/

As a demonstration of the "Worse is Better" in regards to performance, Emery briefly described the progression to modern day performance.

1. Lisp (1960's)
2. Generational garbage collection (1985)
3. Self (1990's) (dynamic compilation, just-in-time compilation)
4. Java's JVM, Javascript's V8 (2000's)

Also mentioned was that part of C's speed was due to its close connection to the hardware it was developed on. `x++` and `++x` were noted to be assembly instructions on the PDP/11.

Emery closed this topic with a diagram of the relative time required to ship software in the MIT and New Jersey styles. You should refer to his PDF for the diagram. In summary: the New Jersey style allows for rapid releases, whereas the MIT style required long periods with little public activity. This results in New Jersey styled software becoming entrenched, and when the MIT style system finally gets released, it has already lost the battle.

11.2 Confinement Problem

Lampson identified the existence of covert information channels (as opposed to overt channels, like the Internet, ttys, and monitors).

These became highly important during the 1990's. Once the large OS makers (read: Microsoft) started to take security seriously, malware makers began to focus solely on covert channels.

Emery mentioned an old timing attack (that has since been "fixed"). Using the time deltas between keystroke, a model of a person's typing could be developed. By recording the time deltas during password entry, the model could predict the user's password. While not fully accurate, the model narrowed the search space by multiple orders of magnitude. The entire process (both building the model and cracking the password) could be done in the space of a few minutes.

Class question: How do we make systems secure?

Answer: Nobody knows (except we do, but using known strategies would make computers worthlessly slow).

Emery made a diagram describing the privacy/integrity (duality? opposition?). Refer to his PDF for the diagram.

11.3 Taint, protecting information

We began discussing the means by which information could (or could not) be secured in a program. The following pseudocode was provided.

```
int secret x = password;     // secret information, do not leak
int s = x;                   // note: s is not secret
print s;                     // x has been leaked
//----
int q = s + 12               // q contains information that can be
```

```
                                  // transformed into x
  //----
  s = read_socket(...)            // read from untrusted source
  system(s)                       // run from untrusted source
```

Perl has amethod for managing these problems called "taint mode". Secret values are tainted, and taint propogates on copy and assignment. Tainted values cannot be output (at all, more on this in a bit). This system is conservative (in the sense we used while talking about garbage collection) and very effectively ensures that tainted values are known.

However, it also has the problem of taint explosion. Taint propogates, which means that the more tainted values there are, the more the taint will spread. Eventually, all values are tainted and cannot be output.

There is a second issue with tainting. If your program interacts with a database, then every value coming out of the database is considered tainted. As you cannot output a tainted value, or an value whose composition contain a tainted value, this data is effectively stuck in your program.

These two issues are resolved through sanitizers (which removes the taint from values) and declassifiers (which allow the partial release of tainted values). Sanitizers and declassifiers are domain specific.

11.4 Information Flow

Taint mode is an example of "Dynamic Information Flow Analysis". That is, it tracks taint at runtime. The alternative approach is "Static Information Flow Analysis", as demonstrated by the JIF (Java + Information Flow) project (which has yet to show that it is worth using).

There is another style, called "Quantitative Information Flow", with a paper by McCamant and Ernst.

Another pseudocode was provided, noting that control flow involving tainted values should taint all values touched by the control flow.

```
if ( is_secret(x[i]) ) {
    a[i] = 1
} else {
    a[i] = 0
}

print a
```

This procedure shows that `a` captures the entirity of the secretness of all values in array `x`. Therefore, `a` should be secret. Tainting control flow will lead to taint explosion. This is called implicit information flow.

Emery described an old timing attack on RSA. Branch statements in the algorithm differed in the number of CPU cycles required to compute each branch. Therefore, by recording the timings, one could rebuild the control flow of the program, and thereby reproduce the secret key. This vulnerability was "fixed" by padding all branches so that they all required exactly the same amount of time to run.

11.5 Philosophy of Computer Security

The above solution is an example of hardening a program against an attack. There are, broadly speaking, two approaches for securing systems.

Fort Knox

Perfect, expensive, nobody gets in or out (for any reason)

Window Locks

Window locks are cheap and provide sufficient barrier to entry. They can be easily bypassed by e.g. smashing the window, but that is loud and invites risk upon the transgressor. This risk discourages the transgressor, and therefore, the window lock discourages the attempt.

The window lock approach tends to be good enough. The prevailing mindset is to make intrusion prohibitively difficult... for whoever you think is targetting you.

11.6 Threat Models

Designing secure system requires an idea of the sort of threats you will face. Threat models can be summarized in two questions: Who is attacking, and what resources can they weild against you? Emery mentioned the "rubberhose attack". This attack vector has an XKCD cartoon: (https://www.xkcd.com/538/)

A number of attack vectors were mentioned:

- physical access (also known as: Game Over)
- social engineering
- phishing
- insider attack / disgruntled employee attack
- supply chain attacks
- rootkits (hypervisors)
- port knocking

11.7 Meltdown and Spectre

Meltdown and Spectre rely on "Flush and Reload" behaviors in the CPU. The intel ASM command `CLFLUSH` was mentioned.

We did not cover Spectre, but did briefly talk about Meltdown. A figure to aid the reader.

```
    +----------+-+-----------------+
x = |          |s|                 |
    +----------+-+-----------------+
       index   k

x[k] = s

x[k] is in kernel memory
```

```
y[x[k] is in user memory
```

By running `y[x[k]]`, the CPU will read in the data and check for authorization at the same time. That is, it will read the data into cache regardless of whether or not the caller has rights to the data. Using a timing attack, the exploit can determine where in cache the memory lives and read it out.

This exploit can read enormous quantities of data per second, making it extremely useful for hostile entities.

In closing, the immortal words of Emery Berger:

“Spectre”