COMPSCI 630 Systems **Fall 2019**

Lecture 13: Fault-Tolerance (Hardware)

Lecturer: Emery Berger *Scribe(s): Kyle Godinho, YangJunqing Qiao*

13.1 Programming Language Security

Hardware was seen as the "Gold Standard" for security. Still, there needed to be ways for programming languages to be more secure and robust against exploits. Some design decisions involve needing to enforce spatial and temporal memory safety, *type* safety, and control-flow integrity. However, these programming language still need to be fast so one cannot sacrifice too much speed in the pursuit of security. Even performance hindrances as much as 5%, in the case of Microsoft, can be seen as a no-go. One such example of a "safe" programming language is *Rust*. Some of the ideas *Rust* implements are:

1. Memory Safety: Tries to provide certain guarantees, unlike *C*, that pointers always point to valid memory addresses.
2. Concurrency
3. Ownership Types: One can have data that is "Owned" and can only be written and read following a certain paradigm. One can *loan* out the ability to read the data and this can be given to however many threads as needed. Just reading data does not allow a chance to modify it. One can also *loan* out the ability to write to the data and this can only be controlled by one thread at a time. When this write ability is *loaned* out it also stops other threads with read ability from reading the data in an effort to stop race conditions.
4. No Garbage Collector
5. "unsafe" sections of code: This is the unfortunate security downfall of *Rust*, however. In trying to get around some of the blocks put in place by the language keeping itself "safe", programmers can declare "unsafe" sections of code that ignore these restrictions. However, that one "unsafe" section can access all of memory and thus this unsafe-ness can be seen as "spreading out", making the whole program unsafe.

13.2 Virtualization

Virtual machines also leverage the concept of sandboxing by restricting their target processes to a specified environment and forcing them to communicate down the system stack through a specified API. Virtualization can be incorporated into the general system STACK (as defined by DARPA) and also pose various security risks. Virtualization software for various programming languages are a common vector of attack.
DARPA tried to create a secure 'STACK' with no memory vulnerabilities however, even such a secure system will encounter high level vulnerabilities. One example is SQL injection attacks. Due to the way that SQL queries are parsed, an attacker can submit an arbitrary query for processing into the system. This does not exploit any memory safety issues and instead takes advantage of high level design choices in the SQL implementation.

13.3 Security

13.3.1 High Level Security

High level security vulnerabilities arise due to the implementation of applications. The SQL query injections outlined in the previous section. Such vulnerabilities are application level security issues and have no general solution. The best that security experts can do is to anticipate or quickly fix known bugs and deploy patches before too much damage can be done. This is commonly referred to as a 'cat and mouse game'.
Another more extreme example is that of a 'rubber hose attack' which assumes the attacker has physical access to crucial security devices (such as threatening the DBA with physical violence).
One also needs to consider what their *Threat Model* is. How powerful an attacker is, what attacks you're expecting to be resistant too, and physical access to what you're trying to protect are examples of things to consider. For example, if one's *Threat Model* includes being resistant to attacks from the external internet but then someone on one's internal network performs the attack and they succeed, that is not something that was planned for.

13.3.2 Low Level Security

These are the security exploits that a secure system stack would seek to prevent. The issues handled here involve spacial, temporal, memory, type safety, and control flow integrity.

1. Drive by attacks: When a user navigates to a malicious website, the site will run plugins and other applets that the browser handles to try and take advantage of zero day vulnerabilities or users who have not updated their software recently. These attacks commonly run known exploits on virtualization software such as the JVM in an attempt to corrupt the stack, deliver their payload, and force the user's machine to execute arbitrary code.

2. Heap spraying: The attacker fills the heap with shell code and NOP sleds and attempts to subvert a single pointer to have their code execute. JIT spraying is a more specific version of this attack. Since JIT compilers can not be run in a no-execute-program environment, malicious code takes advantage of this and sprays the heap with JIT compiled code.

3. Speculative execution leaks (SPECTRE): Attackers would request an arbitrary address to have it stored into memory, locate its address by exploiting speculative execution, and then expose the information using a covert channel. This attack allowed applications in user space to access arbitrary kernel code.

4. Hardware level exploits: The Row hammer exploit used physical principles of how computers are designed to attack their memory. The Row hammer attack depends on the physical phenomenon that a charge can be induced through flipping a bit in RAM; this charge in turn might cause any of its adjacent neighbors to have their values reversed as well. Such an exploit has far reaching consequences. Cloud services depend on the fact that space on hardware can be partitioned into independent sections however, the Row hammer attack negates this assumption as physical storage location is integral to its success. To combat this, cloud service providers such as Amazon physically isolates company secure data on their servers.

13.4 Privacy versus Integrity

Privacy and Integrity can be seen as ”duals” in their function. With Privacy, one is trying to stop data in a safe space from leaving to the unsafe world. With Integrity, one is trying to stop data from the unsafe world from entering your safe space.

13.4.1 Integrity Example

Perl:

1. *Perl's* approach to integrity places focus on data coming in from the world, such as through a network socket, that is considered to be ”tainted”. If trying to just save the data, you would get an error that the data cannot be saved due to it being ”tainted”. In order to save the data, you would need to ”sanitize” the data and in turn remove its labeling as being ”tainted”, therefore making it safe.

13.5 End-to-End Transfer

One method of ensuring a large file gets transferred correctly over a network is through hashing. Before sending, make a hash of the whole file along with hashes for smaller chunks of the file. The receiver, after receiving the whole file, will then check its complete hash. If it's checks out, then they can be pretty positive that there were no errors in receiving the file. If it is wrong, then they can go through checking the chunks of the file they received to the hash of those chunks. This saves time as when they find where the error occurred they can just request only that chunk to be re-sent.