September 19, 2011 SFI and Janus

Ian Fischer

September 25, 2011

1 Homework 1 Discussion

1.1 Enforcement Architecture

Approaches students take to the HW1 enforcement architecture typically fall into three categories:

1. Regular Expressions
2. Shadow Parse
3. Dumbification

Regular Expressions are always breakable. HTML is not a regular language, or even a context-free language.

Shadow Parse refers to the process of taking HTML, generating a parse tree, checking if the parse tree violates the stated policy, and then either emitting the document unchanged if it doesn't violate the policy, or rejecting the document if it does. This fails because the parser used to check the policy is different from the parser used to render the page for the user. The attack is to find a string that parses differently between the the two parsers, which is typically feasible.

Dumbification parses the HTML as above, but then passes the parse tree through some filter that reduces the expressiveness of HTML, removing anything considered dangerous. After removing this, it outputs a new HTML document that should only contain HTML that all reasonable parsers should parse the same way. Properly implemented, this can be secure.

A story to remember the idea of dumbification, from a story by Vernor Vinge. The Universe consists of a nested set of zones where physics moves from increasingly strict in the middle, to increasingly lax at the edge. In the middle, where Earth is, physics restricts what is computable, and prevents the creation of a true Artificial Intelligence. At the edge, on the other hand, AIs exist and are immensely powerful. In fact, even network protocols are so advanced that the packets themselves can be artificially intelligent. An evil AI arises and decides to take over the galaxy. It creates network packets that are intelligent enough to breach the defenses of the other entities at that layer of the Universe, and it begins its conquest. The good guys need to communicate to fight back, but they cannot trust even the network packets they send themselves, since many of their networks are compromised and send out evil AI packets. So they come up with a scheme: they route the packets through the innermost layer of the Universe, which destroys the AI in any packets they send, since that layer of the Universe does not permit AI. Thus, their communication is safe.

A related idea to dummification is paravirtualization. In paravirtualization, the system does not implement some of the more difficult-to-implement instructions of the x86 architecture. It then transforms the input programs so that they only use the easily virtualized instructions. This generally requires modifications to the guest OS.

1.2 Policy Architecture

For the policy architecture, student approaches generally fall under two main categories:

1. Blacklists
2. Whitelists
   (a) Whitelist tags only
   (b) Whitelist tags and attributes

The blacklist approach is essentially always broken. It is too easy to miss some tag or attribute that allows scripts to be attached. The whitelist approach is also broken if only tags are whitelisted. Essentially all tags can have scripts attached to them. Only whitelisting tags and attributes together can be sufficient to prevent attacks.

The character-based taint tracking paper violates these policies and enforcement architectures. They had policy rules that looked like `(<script)`^{t}. They instead should have parsed any tainted HTML and filtered the resulting tainted tree against a whitelist of tags and attributes.

Homework 2 options:

1. Attack other students' homework 1s
2. Do a security code review on real code
3. Do a pen and paper security analysis

After a vote, there was a three-way tie.

2 SFI

(Continued from previous lecture.)

SFI looks like it might be useless. Once the sandboxed process has been started, there doesn't seem to be any way to get a result back out – all memory operations and jump targets are confined to the sandbox.

2.1 How do we let SFI return content to the browser?

- We could allow a page of execute-only memory in the SFI address space that the sandboxed code could jump to that could subsequently return to browser code.
- We could have a known memory region for the browser to read from, but we still need to transfer control back to the browser.
- We could allow the sandboxed process to jump to exactly one address outside of the sandbox that is statically checked for at compile-time.

2.2 How do we extend SFI to allow a plugin to call APIs?

- We could have a set of allowable jump targets with known memory addresses for the arguments.
- We could have one jump target with an extra parameter for the function name.

- Watch out! You can't jump, you have to use the stack. You have to watch for stack forgery.
- Watch out! The plugin could cause stack exhaustion (controlling precisely how far execution gets before the program halts, which can result in the program leaving things in a dirty state). You have to switch stacks to prevent this.
- Watch out! Reading data from memory for the arguments is dangerous. The plugin could be multithreaded, so the data could change under you. This is called a *Time Of Check To Time Of Use* (TOCTTOU) bug.

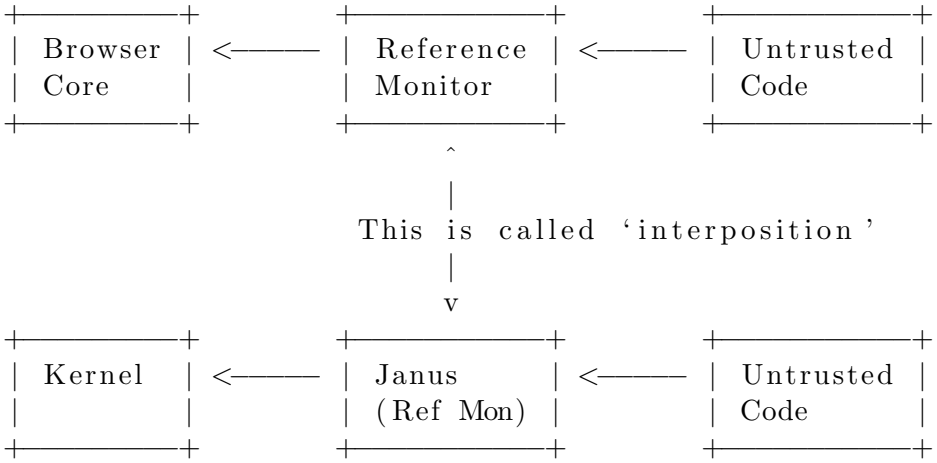
2.3 Time Of Check To Time Of Use

```
void f (void *p) {
  if (ok(p)) {
    do_it(p);
  }
}
```

There is a TOCTTOU bug between lines 2 and 3 above. We can fix it by copying the data before performing the check. More generally, TOCTTOU bugs occur because of global, modifiable state being shared between multiple concurrent processes. Thus, to fix TOCTTOU bugs, you need to make the state in question either private or unmodifiable. In this case, copying the data before the check makes it private to the current thread.

Note that all of these issues with SFI occur when you are trying to write a kernel as well.

2.4 High-Level SFI



Interpostion is the security version of adding a layer of indirection to solve a problem.

3 Why does Janus suck?

Why did we read Janus? We now know many of the things that Janus got wrong. Also, it was a class project, so it gives an idea of the scope a class project can take.

Some of the problems with Janus:

1. Concurrency

2. Shadow State
3. OS-dependent
4. Defining Security Policies
5. Static Security Policies
6. TOCTTOU Vulnerabilities

Concurrency is a problem for Janus because Janus depends on the state of the kernel. To allow concurrency, there would need to be one Janus process per sandbox process. However, Janus depends on state changes from previous system calls, so it would be possible to have one process do one system call that is apparently safe, and another do another apparently safe call, but the two calls together violate the policy of one or both processes.

Shadow state is the copy of the kernel state that Janus has to maintain in its own address space, since it is a user-level process. If Janus ever fails to maintain the same state as the kernel, there is the possibility of security vulnerabilities. For example, in the original implementation, they failed to correctly maintain the state of the `fcntl(DUP_FD, fd1, fd2)` call, which exposed a vulnerability. This problem could be fixed by implementing Janus in the kernel, by updating the kernel to allow state inspection, or by making the system call interface stateless, but all of these solutions obviously require modifying the kernel.

The implementation of Janus was very *OS-dependent*. This is because different operating systems expose different system call interfaces. Some kernels required hooking all system calls if any are hooked at all, which slows things down substantially. Other kernels have too many system calls for it to be easy to reason about what is a good policy and to hook all the appropriate calls.

Defining security policies themselves is also quite difficult, and it is unclear who should have the responsibility. Manually generating security policies that make sense is difficult. Sysadmins and users aren't going to do it. That leaves OS vendors and the program authors themselves. You could try to dynamically learn the required system calls by watching a program execution. However, there may be program executions that you don't detect – i.e., this has potential coverage issues. It would also be necessary to generalize for the arguments to given system calls, but to avoid overgeneralizing as well. For example, if a program opens a different temporary file on each execution, you have to correctly learn that it is only opening files in the /tmp folder, rather than that it is opening an arbitrary file. Also, sometimes programs try to do things that they shouldn't, which a correct policy should prevent, but a learned policy would not. Instead, you could try to statically analyze the code to see which calls it uses. However, the only improvement this gives over dynamic analysis is that it reduces coverage problems. In general, static policies suffer from the fact that it is often easy to find a sequence of system calls to do something that causes harm, and return-oriented programming may allow an attacker to create such a sequence given an exploitable vulnerability somewhere in the code.

Static security policies are generally an issue, then. It seems that policies need to be dynamic to prevent overprivileging. We did not discuss how you might do this, however.

TOCTTOU vulnerabilities are particularly difficult to avoid. For example, if a policy states that `exec` can only be called on non-`setuid` programs, it is still possible for a given program that is about to be `exec`'d to have its privileges increased with a `setuid` call from another thread or process.

Challenge: Think how to extend Janus so the policy engine allows things like `open("/a/b/c/d", O_RDONLY)` if each component in the path is not a symlink, in the presence of concurrency.