In last class we saw how to sandbox untrusted code, but what if we need to run the untrusted code within the same address space of trusted program in order to avoid overhead of context switches?

We can run native code in the browser. We want to break a large software in smaller modules because one module doesn’t trust the other module. Like BSD, Netfilter, Packet filter rules, packet sniffer etc. A packet sniffer is used to monitor network. If we run it too frequently, on a heavy network, it may be overflown with data. So say the packet filter puts stipulations on the kind of data (limiting the packet length). But there is a problem here because the after the filtering is done, it is in the kernel space and then it is passed on the userspace.

So the Big question is running untrusted code in a trusted program. We would need separation of privileges. Read and write apps are kept separate and they can communicate in some way.

Chrome and its plugins
Plugins are run in a separate process and hence trusting the OS and hardware for isolation. The downside being overhead like context switching, replication of data etc.

In general running two different modules in separate process is okay when communicating messages are not large. Otherwise it’s a waste. So we delve deep into understanding how Native client puts mutually distrusting modules into the same address space.

**Inline Reference Monitors and Native Client (& software Fault Isolation & others)**

The problems we need to solve outright
- Memory isolation: restrict untrusted code to r/w only its own memory. We don’t want the untrusted code to open random files

```
<table>
<thead>
<tr>
<th>Trusted region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untrusted region</td>
</tr>
<tr>
<td>Trusted region</td>
</tr>
</tbody>
</table>
```

- Controlling privileged actions like system calls, instructions, etc. This might look like we are reinventing the wheel, but No. The tradeoff by avoiding context switching for just a little extra overhead makes sense. A more file grained approach where parts of a program has a different levels of privilege.

- Cross domain calls: well if the untrusted code cannot modify any external memory and cannot make any syscalls, what’s the use of the code? We need to have the untrusted component to access the trusted component in some restricted way. Like allow the untrusted comp to call something like a safe_open() instead of open().

**Solution**

1. Insert runtime checks into the untrusted binary for memory safety
2. Prevent bad jumps for something like

```
Jump
```

check bounds on memory access

```
store %r0, [%r1]
```

3. Verifying correct compilation using a NACL verifier. Only if the untrusted .o OK verified, the code can run into the untrusted region.

![Diagram of code production and verification process]

**Intelligence of verifier**

We define new rule for verifications

```
%r14 – lower bound
%r15 – upper bound
```

So in order to restrict and make the load, safe we do

```
b1 %r0, %r14, .Err
b1 %r15, %r0, .Err
load [%r0], %r1
```

This can be optimized as we are using 2 registers for comparing, branch, comparing, branch.

**Optimizing bounds** (like baggybounds)

```
2^k
```

![Diagram of program counter and trusted/untrusted regions]
We can make use of Program Counter and do Pointer swizzling to force the load address of the untrusted code to be within the bounds of the untrusted region.

\[ \ldots k \ldots \]

\[
\begin{array}{l}
\text{shr} \ %\text{pc}, k, \%r2 \\
\text{shl} \ %r2, k, \%r2 \\
\text{shl} \ %r0, 64 - k, \%r0 \\
\text{shr} \ %r0, 64 - k, \%r0 \\
\text{load} \ [%r0], \%r1
\end{array}
\]

**How to prevent jumps**

1. Absolute jumps like
   \[
   \text{jmp} \ 0x12345
   \]
   can be checked at compilation for it to be within bounds

2. Position independent code jumps
   \[
   \text{jmp} \ %\text{pc} + \text{OFFSET}
   \]
   where offset is a constant, can be checked if we can disassemble the entire code
   a. bounds\text{check} \ %r0
   b. check that \ %r0 = 0 \ mod \ 32

3. for jumps like
   \[
   \text{jmp} \ %r0
   \]
   We can’t check it statistically. So a solution is to maintain 32 byte boundaries for every set of instructions.

**Verifier ensures:**

1. all jmps 0 mod 32
2. all jmps are bound checked
3. all load/store bound check
4. No instruction crosses the 32-byte boundary
5. checks must be in the same block as the inst they guard
6. No sys calls

So with this we achieve all the requirements for Inline Reference Monitors & Native Client except cross-domain calls.

----------------------------------