CSE409, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, October 31st, 2011

Google's native client project

Introduction and overview

Last time we discussed sandboxing, and we agreed that:

- If you trust the program, you get the policy from the program
- If you don't, you or the OS define the policy for the program

The purpose of the **Google Native Client** is to run x86 code in your browser *safely*, such that the code cannot affect your system.

General principle

We're going to have a process, with its virtual address space, and inside it there will be an **untrusted module** (the plugin's x86 code). The bottom line there will be a lot of problems to deal with, since we are **running code we don't trust inside the address space of code we do trust**.

Question: Why embed the module in the process like this?
Answer: Because it's necessary for good performance, since the module needs to communicate with the browser process a lot. We could have the module have its own different process, but IPC is very expensive. As we'll see, running the code inside the process also enables safe sandboxing.

Questions: Should you have more than one module in a sandbox? What are the interaction rules? What if one of the modules is buggy?

- We must isolate untrusted code even if it is buggy
- Trusted code can call untrusted code, and vice-versa
- Untrusted code cannot call untrusted code directly; it has to go through the trusted code

Examples of **native client** code **applications**: browser extensions, driver code, database functions, packet filters

Dangers to avoid

Applications running inside the native client should **never be able to do the following** things:

- Crash computer
- Resource exhaustion
- Arbitrary memory writes (or reads)
- System calls
- Forbidden x86 instructions
- Unconstrained control transfers (like jumping anywhere inside a trusted function)
    - Solved using "call gates," a function that untrusted code can call in the trusted code

Insight: It turns out, that the **main issue** we need to worry about are **memory writes/reads** and **forbidden instructions**. Once we solve these two, the rest will follow.

On the other hand, native client applications need to be allowed to perform certain *risky* operations:

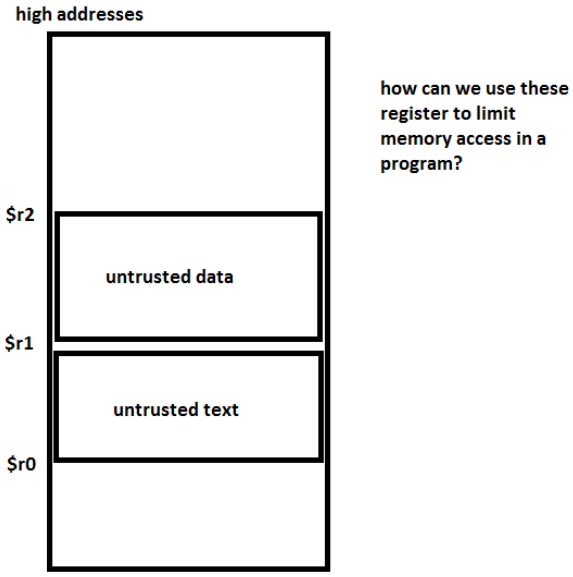
- Reads and writes to special memory area
- Calls into trusted code

- Could lead to exploitation
- Imagine the following `secureOpen` function

```
if(policy.allow(args)) {
    open(args); //  what if an attacker jumps here directly?
}
```

How to limit memory access

Question: If you had control over the compiler and loader, how would you modify them to ensure legal memory access?



For instance, consider the following instruction: `ld $r4, [$r5];` The **compiler** should ensure the program is only reading the data section, with a check like the following:

```
//  Ensures the address in $r5 points within the untrusted data section
if($r5 < $r1 || $r2 <= $r5)
    abort();
```

The **loader** should confirm that:

- all load/store instructions are preceded by checks like the one above
- no instructions modify the `$r0, $r1, $r2` registers, since they store the memory bounds

Insight: Your security can only depend on the loader.

Question: What if an attacker just jumps in his assembly code to a load instruction so that the checking is not made?

We want to limit where the program can jump. A **basic block** is a sequence of instructions that contains no branches and ends with a branch. If we **constrain jumps** to always be in the **beginning of a basic block**, and we also make sure that every basic block starts with proper checking code, then a native client program will never be able to jump past the checking.

To handle jumps, Google's native client divides the code section (.text) **into 32-byte chunks**. The loader ensures that every jump is at the beginning of one of these chunks. Also, if there is a load instruction inside a chunk, the loader will check to see that it is preceded by a check in the same chunk.

Checking is simple. The loader can check that the address is at the beginning of a 32-byte chunk:

```
if($r4 & 0x1f)
      abort();
```

Doing such checks costs around 3 or more instructions. Can we do it faster, possibly crashing the program? Yes!

```
and $r4, $r4, 0xffffffe0
```

Example: We can:

- assume the size of the untrusted .text size is a power of 2
- assume all the addresses in the .text section begin with the exact same 3 hex digits
    - if more/less space it's needed, adjust the number of hex digits above

To ensure that every jump instruction always jumps to the beginning of a 32-byte chunk, we can simply remove some of the last bits in every address passed to the jump instructions.

```
and $r4, $r4, 0x000fffe0
or  $r4, $r4, 0xabc00000
jmp $r4
```

In practice, we'll do an `or $r4, $r4, $r0`, since the .text section starts at `$r0`

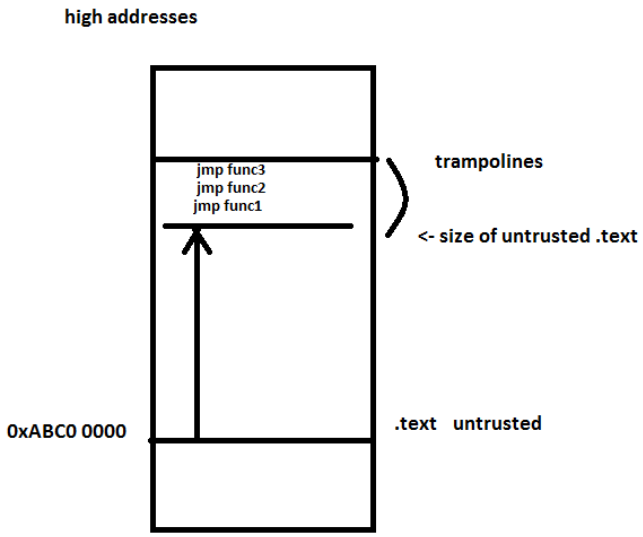
Complications

What about relative jumps like `jmp $pc + 24`? The loader can check statically, at load-time, that such "constant" jumps are within the .text section.

What about returns, which jump to an address of the stack? You just have to be check them with the same **and/or** method above.

Invoking functions inside the trusted code

In order to invoke trusted code, we need **call gates** – a way to allow someone to make a function call in a controlled way.



The loader can place **trampolines** in the remaining space inside the .text section (or just allocate more space).

Now we can make the untrusted code jump to the trampoline, which in turn jumps to the trusted .text section. You can prevent certain functions from ever being called by not setting trampolines for them.

How come trampolines allowed to jump to trusted code? They are trusted, since the loader put them there.

What about the stack?

We can put the stack inside the .data section. **On a trusted function call**, trusted code must:

- save the state of the untrusted module (the stack pointer, program counter)
- copy the arguments to the trusted stack
- call real trusted function
- copy result into untrusted stack
- restore state
- return control to untrusted module.
    - You must make sure you jump back to a 32-byte aligned address, by maybe padding the chunk with no-ops if the jump was made from the middle of the chunk.