CSE 509 Lecture 22

Prof. Rob Johnson, Scribe:Arun Rathakrishnan

November 18, 2013

1 Native Client Applications

Native Client is a mechanism for an application to run untrusted code in its address space. There are several situtations in which untrusted code has to be executed by an application. Network Packet filters require the kernel execute some untrusted code in the kernel space. Plugins supported by web browsers also fall in this category. In Google Chrome, the plugin is executed as a separate process so that, the browser process is isolated from the untrusted plugin code. The aim is to protect the untrusted code from adversely affecting the target application and prevent it from stealing sensitive information.

The isolated processes communicate through Inter Process Communication. But if the interaction between processes is high, there is a huge communication overhead due to IPC. This overhead can be avoided, if untrusted code runs in the same address space as the application,

- but is prevented from accessing some parts of the application's resources such as code section, stack, heap, etc.
- The native client must not be allowed to make arbitrary system calls.

. If we can enforce the above restrictions, the untrusted code can execute in the same process as the application reducing the communication costs due to IPC.

2 Inline Reference Monitors

When native code executes in the target application, the below conditions must hold.

Memory Isolation Restrict untrusted component to only read or write its own memory.

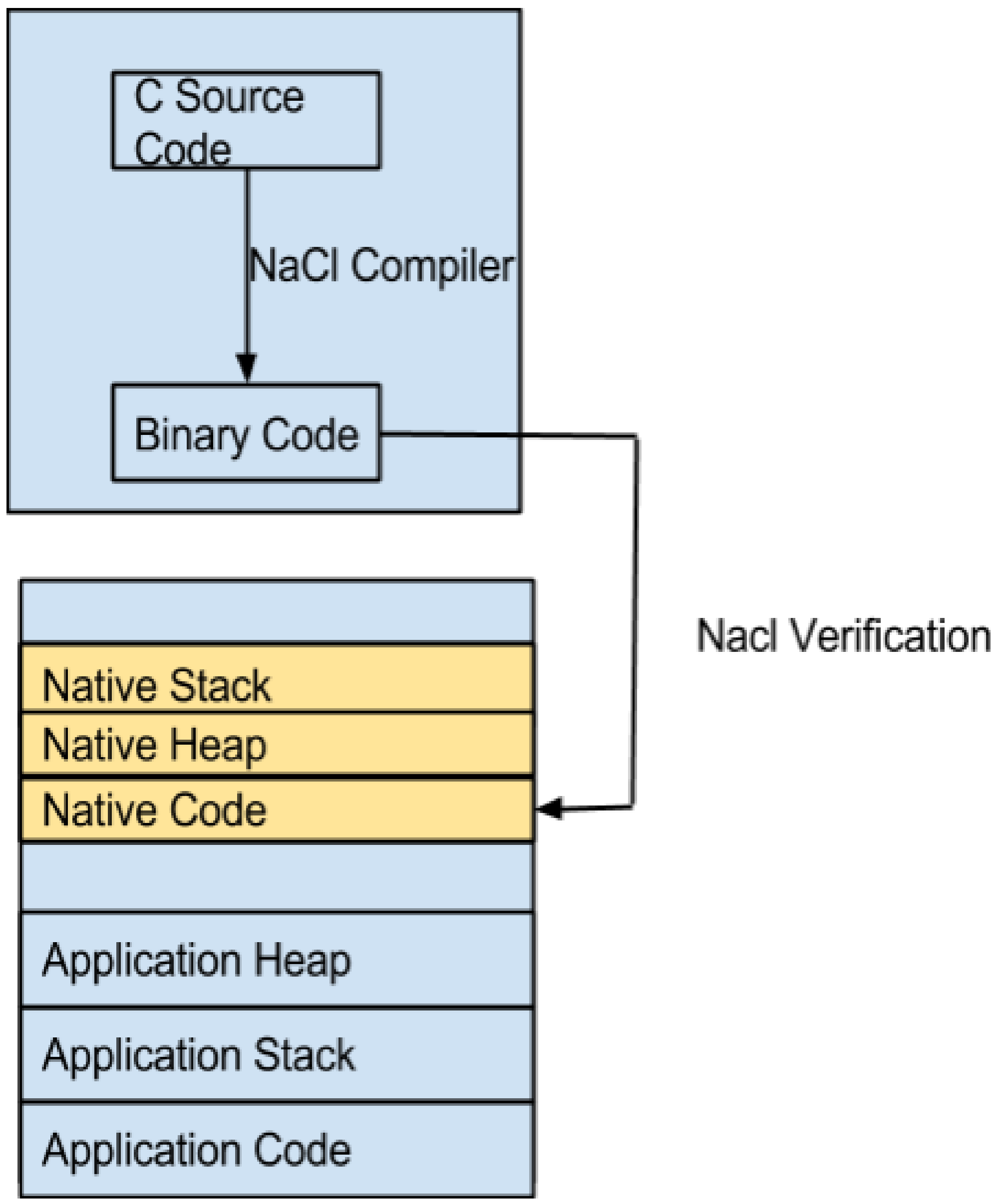
Control Privileged Actions Prevent system calls and ensure the code runs in user mode.

Cross Domain Calls Communicate with the target application. Application needs to make calls to the native code and native code may make use of libraries that are part of the application's address space.

A compiler for native clients can perform binary instrumentation by,

- Inserting run-time checks into the untrusted binary. Check if memory references are safe.
- Preventing bad jumps, that bypass these checks.
- Verify correctness of the compilation.

2.1 Verification Process



Compilation, Verification and Memory Organization

2.2 Organization

A separate compiler is required for compiling applications that support native clients. The untrusted code is confined to a separate memory region within the process address space that supports stack, heap and code section that are isolated from the application's. When the control goes to native code, we track all memory references and ensure that they are within the bounds of the isolated memory region.

2.3 Bounds Checking

One strategy is to use two registers to store the bounds of the isolated memory region. Before any memory access we need to check if the address is within bounds. In the example below, we consider the execution of a load instruction in a 64 bit machine.

```
load [%r0], %r1

bl %r0, %r14, .ERR
bl %r15, %r0, .ERR
```

By aligning the isolated memory region to power of 2 boundaries, we can reduce the number of registers used or the number of instructions required for the check. When the control is with the untrusted code PC value will always remain within the 2^k range. By making use of the fact we know k and can force any memory access to be within the untrusted region.

```
shr %pc, k, %r2
shl %r0, 64-k, %r0
shr %r0, 64-k, %r0
or %r2, %r1
```

Even if the address is outside the range, it is forced to be within the memory range. This is called pointer swizzling. Since we only care about confining the memory access in the untrusted region, this is a valid approach.

2.4 Preventing Bad Jumps

Once the bounds checking instructions are in place, we must ensure that the native binary does not have a way to get around this. The only way to prevent the execution of such instructions is to insert jump instructions so that the machine jumps over bounds checking code and performs illegal memory accesses.

Jump instructions come in three flavours.

- jmp direct address
- jmp %pc + offset
- jmp %r0

The first two types can be verified by static checking to see if they skip bounds checking. The third type of jump occurs during in switch case statements and can be resolved only during the runtime. This is performed by the same bounds check on address in register's contents. But this alone is not sufficient to ensure that no bounds instruction is skipped.

However there is a simple trick that is sufficient. We organize the binary instructions, so that the bounds check code always start at 32 byte boundaries. As result, if the original instruction to be executed after jump is a memory access we are sure that bounds checking instructions will be at the next instruction to which the control transfers to.

Given the native binary, we must disassemble the binary to get byte code, which is later instrumented with boundary conditions, that start at 32 byte boundaries. The instrumented code must satisfy the following conditions to prevent illegal access by the untrusted code.

- All jumps are to 0 mod 32.
- All jumps are bounds checked.
- All loads/stores are bounds checked.
- No instruction stradles the 32 byte boundary. We can pad using nops to achieve this.
- Bound check instruction must be within the same 32 byte range as the memory access instruction and must begin at the boundary of the region.

A linear sweep disassembler can convert the native application binary to byte code, that can be executed within the address space of the target application.