Google Native Client

Analysis Of A Secure Browser Plugin Sandbox

Chris Rohlf
Leaf SR
leafsr.com

Introduction

NaCl (Native Client) is Google's attempt at bringing millions of lines of existing C/C++ code to the Chrome web browser. NaCl allows you to run native applications within Chrome that range from PDF readers to 3D games from the safety of a sandbox with all the performance users expect from native code. For this reason it is often mistaken as a reimplementation of Microsoft's ActiveX technology. Nothing could be further from the truth. NaCl attempts to bring a much needed layer of security to browser plugins.

Competing browser plugin technologies such as Microsoft's ActiveX and NPAPI (Netscape Plugin Application Programming Interface), a standard browser plugin API supported by most browsers, rely on click through dialogs to allow or disallow a particular plugin from executing. This design essentially relies on the user to differentiate between safe and malicious plugins. This model is inherently flawed and users have suffered the consequences since its inception. NaCl assumes all modules are malicious and enforces the same security mechanisms on all of them.

Google has designed NaCl such that multiple exploitable conditions are required in order for an attacker to break free from its sandbox container but at the same time provide much higher performance than a JavaScript interpreter or JIT engine. Not only does NaCl provide a higher performance by comparison but it also brings other languages, such as Ruby, to client side web applications.

NEXE modules are currently only distributed through the Chrome web store. This may change in the future as the technology matures. NaCl's market share is still small but may soon be changing due to the fact it is included by default with Chrome since version 14. The research the NaCl team has performed into software fault isolation and sandboxing in general will undoubtedly influence future program isolation and software security models with similar goals.

Executive Summary

The inner workings of NaCl can appear large, complex and intimidating at first. The entire architecture is too large to be covered in this document. So we attempt to explain the security relevant components, their attack surface, where security vulnerabilities have been found and how NaCl fits into the overall Chrome security architecture.

In June of 2011 while employed at Matasano Security the author of this paper performed a source code review of the NaCl PPAPI (Pepper Plugin Application Programming Interface) interfaces for Google before NaCl was included in Chrome. This review resulted in 10 previously unknown security vulnerabilities. These vulnerabilities are briefly covered in this paper along with an overview of the exposed interfaces they were discovered in.

In January of 2012 the author of this paper and another security researcher, Cody Brocious, developed a fuzzer for the NaCl PPAPI interfaces for Google. This fuzzer took as its inputs the NaCl PPAPI IDL (Interface Description Language) files found in the Chrome source tree and output a NEXE module in C++ that fuzzes those interfaces. This fuzzer is named Chrome-Shaker. A section of this document details the approach and implementation of this tool as well as the challenges faced in developing it.

There are many mentions of the Chrome sandbox throughout this document. Despite the major role it plays in containing NaCl components the details of its implementation are beyond the scope of this research document. For more information on how the Chrome sandbox works please see the section labeled references.

Authors Note: Google neither paid for nor endorses this research paper. However they were given advanced notice of its publication.

NaCl Architecture

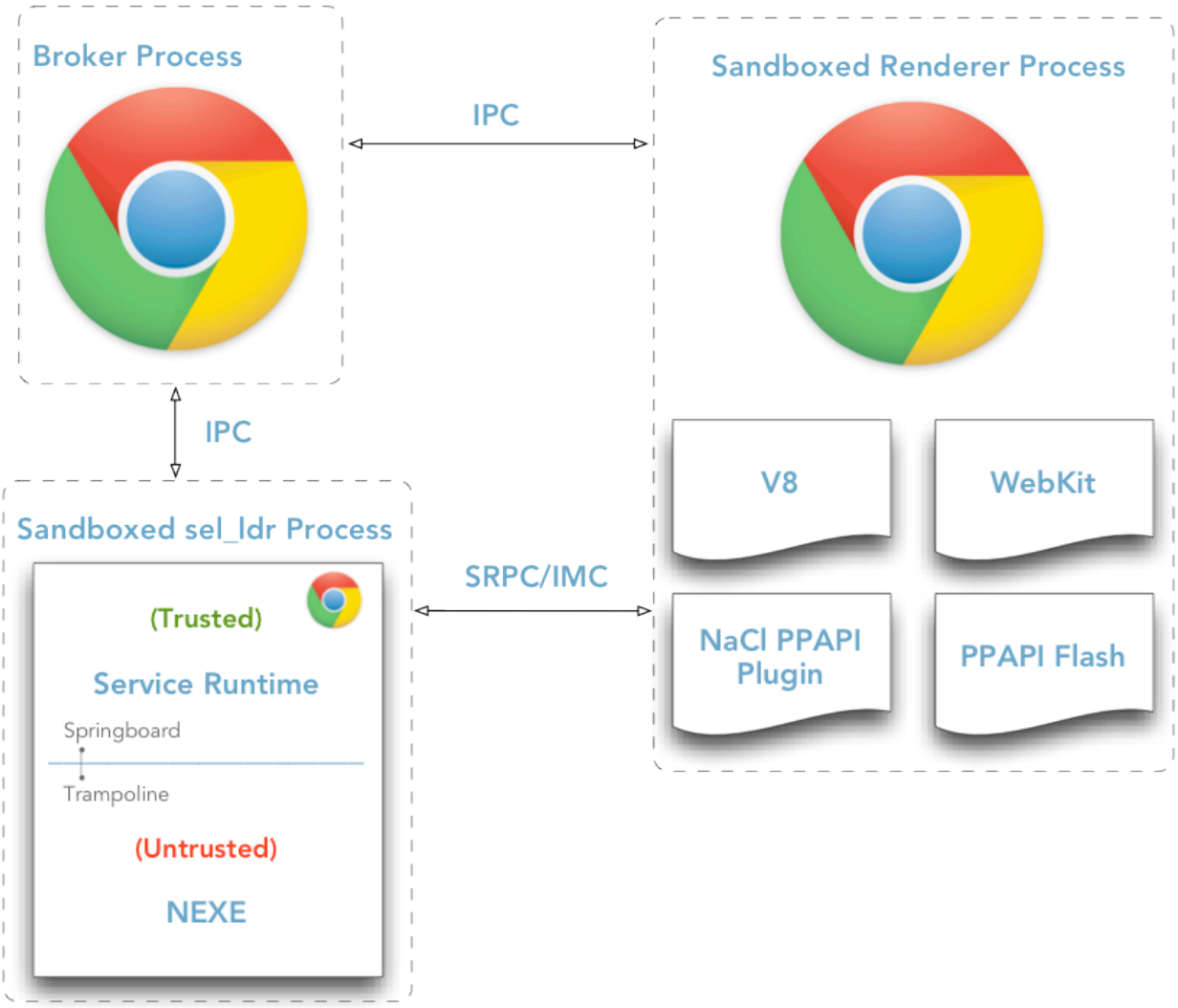
NaCl is a collection of components that includes an SDK with a modified GCC toolchain, a PPAPI browser plugin, a NEXE (Native Client Executable) loader and an SRPC (Simple Remote Procedure Call) messaging system. Each one of these components plays a vital role in compiling, loading and ultimately executing the NEXE inside a trusted sandbox.

NaCl is unique in that it gives untrusted NEXE modules raw access to the CPU while denying access to the rest of the system. It is often compared to Microsoft's ActiveX stack or traditional native code browser plugins. While ActiveX was designed for more than just browser plugins that is where it is primarily used. ActiveX modules have raw CPU access but are completely unrestricted in what API or system calls they can make. NPAPI plugins are very similar to ActiveX in that the plugin runs unrestricted within the context of the browser. They too can access resources the browser has rights to.

The NaCl architecture is best described as split between its trusted and untrusted components. Trusted components are defined as code that a NEXE module has no access to and cannot execute directly. Conversely untrusted, but validated, code is under direct influence by the NEXE module and may be executed at any time. The line between these two will become clearer as each security relevant portion of the architecture is explained.

NaCl currently works on Intel 32 bit x86, Intel x86_64 and ARM processors. Because NEXE modules are given raw access to the CPU, low level software and hardware security boundaries must be utilized. These security boundaries are enforced differently on each supported platform due to specific architectural designs. This document explicitly focuses on the Intel 32 bit x86 architecture with a Windows operating system unless noted otherwise.

At a very high level C/C++ code is compiled and placed in the ELF (Executable Linkable Format) file format, loaded and validated by the service runtime and then communicates with the NaCl browser plugin via SRPC/IMC (Inter-Module Communication) protocols. JavaScript loaded from the same origin as the NEXE can communicate with the module via the `PostMessage` function. In the next few sections we break down each of these components and explain in detail how they work.



NaCl documentation often refers to an inner sandbox and an outer sandbox. The inner sandbox is the software fault isolation enforced in the service runtime. The outer sandbox is the sandbox provided by Chrome. It is important to note that if a NEXE module can execute arbitrary code within a sandboxed renderer or the service runtime this is considered a privilege escalation even though the execution context is still contained within the outer sandbox. **If a NEXE module can execute instructions that were not validated by the service runtime then the security provided by Native Client is broken.**

Note: A lot has changed in NaCl since the release of Google's original research paper. Much of this research was performed by reading NaCl code and what up to date documentation can be found online. The main directories in the Chrome source tree where NaCl code can be found are shown below:

```
$CHROME/src/chrome/nacl
$CHROME/src/native_client
$CHROME/src/ppapi/native_client
```

The back end PPAPI implementation can be found at:

```
$CHROME/src/webkit/plugins/ppapi
```

NEXE Modules From CTOR to DTOR

NEXE modules are loaded when a user browses to an HTML page that contains an `embed` element similar to what is shown below:

```
    <embed name="NaCl_module"
           id="hello_world"
           width=200 height=200
           src="hello_world.nmf"
           type="application/x-NaCl" />
```

The `src` attribute of the `embed` element specifies where the NEXE manifest file can be found. Manifest files contain simple JSON structures that specify the name and location of the NEXE executable, associated libraries and the architecture they were compiled for. The build process scripts included with the SDK automatically generate the files at build time. Below is an example manifest file for the example 32 bit 'hello-world' application included in the SDK.

```
{ "files": {
    "libgcc_s.so.1": { "x86-32": { "url": "lib32/libgcc_s.so.1" } },
    "main.nexe": { "x86-32": { "url": "hw.nexe" } },
    "libc.so.3c8d1f2e": { "x86-32": { "url": "lib32/libc.so.3c8d1f2e" } },
    "libpthread.so.3c8d1f2e": { "x86-32": { "url": "lib32/libpthread.so.
3c8d1f2e" } } },
  "program": { "x86-32": { "url": "lib32/runnable-ld.so" } }
}
```

The NaCl plugin opens and parses the manifest file's JSON. Once the location of the NEXE is known the NaCl plugin uses existing PPAPI interfaces to download the NEXE file. This is accomplished by asking the browser to either stream the NEXE to a file or to an internal buffer. Once the file has been successfully retrieved the plugin asks the Chrome browser to start the service runtime and passes to it a file descriptor for the NEXE module.

Depending on the platform the service runtime is either a stand alone executable or statically compiled into the Chrome browser itself. A stand alone executable is required on 64 bit Windows because the service runtime must be 32 bit. In that situation a stand alone service runtime is launched named `nacl64.exe`. For the purposes of this paper we assume the service runtime is compiled into a 32 bit Chrome executable. The browser process launches a new process that will be known as the service runtime. The service runtime is protected by the same sandbox as the Chrome renderer process. In NaCl this is referred to as the 'outer sandbox' as it will enforce strict policies on what resources even the trusted side of the service runtime may access.

NEXE modules must be compiled using a modified GCC toolchain that ships with the NaCl SDK. This modified toolchain ensures that the resulting NEXE object code contains the proper ELF headers and conforms to a strict instruction alignment. We start by taking the `hello_world_glibc` example from the *pepper 18* version of the NaCl SDK, compile it as `hw.nexe` and examine it using the `readelf` utility and a disassembler.

NEXE modules are shipped as 32 bit ELF binaries, an open executable format that is also used by Linux. ELF can be fairly complex, but in general the file is broken down into a number of headers or segments. Each header contains offsets to different areas of the file, for example where other headers, debug symbols, relocation or instructions can be found. A NEXE module contains all the typical ELF sections required for loading it into memory: the ELF Header, Program Headers, Section Headers and the Dynamic Segment if its dynamically linked.

The two main validation functions are `NaClElfImageValidateElfHeader` and `NaClElfImageValidateProgramHeaders`. The former is straight forward and small, but the ELF program headers are a map to where all untrusted instructions and data can be found in the executable. The process of validating the program headers is more involved and includes validating segment load addresses and sizes. NaCl also imposes additional rules such as the NEXE can have no more than one executable segment and that the `PT_GNUSTACK` segment does not have executable permissions. The service runtime loads each segment after the program headers have been validated. We can locate the executable instructions in the NEXE by extracting the ELF program headers and looking for the executable segment using the `readelf` command. This segment is shown in bold below:

```
$ readelf -l hw.nexe
Elf file type is EXEC (Executable file)
Entry point 0x10004a0
There are 8 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  INTERP         0x000ee0 0x11000ee0 0x11000ee0 0x00019 0x00019 R   0x1
      [Requesting program interpreter: /lib/ld-NaCl-x86-32.so.1]
  LOAD           0x000140 0x01000140 0x01000140 0x00da0 0x00da0 R E 0x10000
  LOAD           0x000ee0 0x11000ee0 0x11000ee0 0x00478 0x00478 R   0x10000
  LOAD           0x001358 0x11011358 0x11011358 0x0013c 0x0013c RW  0x10000
  LOAD           0x010000 0x11020000 0x11020000 0x00000 0x0002c RW  0x10000
  DYNAMIC        0x00136c 0x1101136c 0x1101136c 0x000d8 0x000d8 RW  0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  TLS            0x000000 0x00000000 0x00000000 0x00000 0x00000 R   0x4
```

Using IDA Pro we can examine this code segment. In order to ensure that only validated code will ever be executed all instructions in a NEXE module must reside in 32 byte aligned chunks of memory. Instructions may never straddle these boundaries. If a block of code exceeds this size then a branch instruction or a series of NOP instructions is used to transfer control to the next instruction at the start of the next 32 byte aligned chunk. The `PpapiPluginStart` function from `hw.nexe` is shown partially disassembled below. Note the 32 byte instruction alignment and the `NOP` instruction padding.

```
01000ac0 <PpapiPluginStart>:
 1000ac0:       53                      push   %ebx
 1000ac1:       83 ec 28                sub    $0x28,%esp
 1000ac4:       a1 00 00 02 11          mov    0x11020000,%eax
 1000ac9:       8b 08                   mov    (%eax),%ecx
 1000acb:       85 c9                   test   %ecx,%ecx
 1000acd:       74 31                   je     1000b00 <PpapiPluginStart+0x40>
 1000acf:       eb 0f                   jmp    1000ae0 <PpapiPluginStart+0x20>
 1000ad1:       90                      nop
 1000ad2:       90                      nop
 1000ad3:       90                      nop
 1000ad4:       90                      nop
 1000ad5:       90                      nop
 1000ad6:       90                      nop
 1000ad7:       90                      nop
 1000ad8:       90                      nop
 1000ad9:       90                      nop
 1000ada:       90                      nop
 1000adb:       90                      nop
 1000adc:       90                      nop
 1000add:       90                      nop
 1000ade:       90                      nop
 1000adf:       90                      nop
 1000ae0:       83 c0 04                add    $0x4,%eax
 1000ae3:       8b 10                   mov    (%eax),%edx
 1000ae5:       85 d2                   test   %edx,%edx
 1000ae7:       75 f7                   jne    1000ae0 <PpapiPluginStart+0x20>
```

```
1000ae9:       eb 15                   jmp    1000b00 <PpapiPluginStart+0x40>
1000aeb:       90                      nop
```

Instruction alignment is critical to the security of NaCl on the x86 platform. This is because there is no mandatory instruction alignment enforced by the processor at runtime. On the x86 architecture it is perfectly legal for a branch instruction to jump into the middle of another instruction sequence and begin executing instructions. This is precisely how ROP (Return Oriented Programming) techniques work, by chaining together unintended code sequences found within the existing executable segment of the process. Allowing this behavior in NaCl would allow untrusted NEXE modules to execute unverified code and escape the inner sandbox.

The modified GCC compiler produces a NEXE module that conforms to the rules of the NaCl system at compile time. Of course an attacker can modify these binaries after compilation so NEXE modules are put through strict validation routines at runtime that include disassembly of every instruction contained in the program.

After the ELF structures have been loaded the individual instructions must be validated before execution can begin. The instruction validator is a disassembler for each supported instruction set. Because untrusted x86 NEXE instructions are always mapped at predictable 32 byte boundaries, validation of instructions can be trusted by always disassembling from the proper start address and ensuring all branch instruction targets are at properly aligned addresses. That is to say as long as the NEXE module cannot execute code at arbitrary addresses we can safely trust the disassembler to validate instructions given a safe starting point. The service runtime maintains a list of x86 instructions it considers safe and a list of those it considers unsafe. Each are marked with the `NaCli_` prefix. Instructions marked `NaCli_ILLEGAL` will cause the validator to throw an error and reject the NEXE module. Banned instructions include those that execute system calls directly such as `sysenter`, or those that modify segment registers such as `lcall`.

Function pointers and other indirect control flow transfers are handled specially in NaCl. For example a virtual function pointer may first be loaded into a register such as `%eax` and then followed by the `call %eax` instruction. This is supported by performing alignment masking on the register before invoking the `call`. This will ensure the resulting value in `%eax` points at a 32 byte aligned chunk of instructions that has been processed by the validator. This is to prevent malicious NEXE modules, or an attacker exploiting a vulnerability in a NEXE module from executing arbitrary instructions. An example of this is shown below:

```
0x100057b:       83 e0 e0        and    $0xfffffe0,%eax   ; eax = 0x100057a
0x100057e:       ff d0           call   *%eax             ; eax = 0x1000560
```

Another special case is the `ret` instruction, which is illegal and will cause a NEXE module to be rejected. Untrusted code could change a saved return address on the stack and execute a `ret` to transfer control to a ROP chain formed from existing instructions. Instead of a `ret` instruction functions are exited via an indirect `jmp` and manually resetting the stack.

The NaCL SDK includes a stand alone validator tool developers can run against their compiled NEXE modules. The validator can be found at the following path:

```
nacl_sdk/pepper_X/tools/ncval_x86_32
```

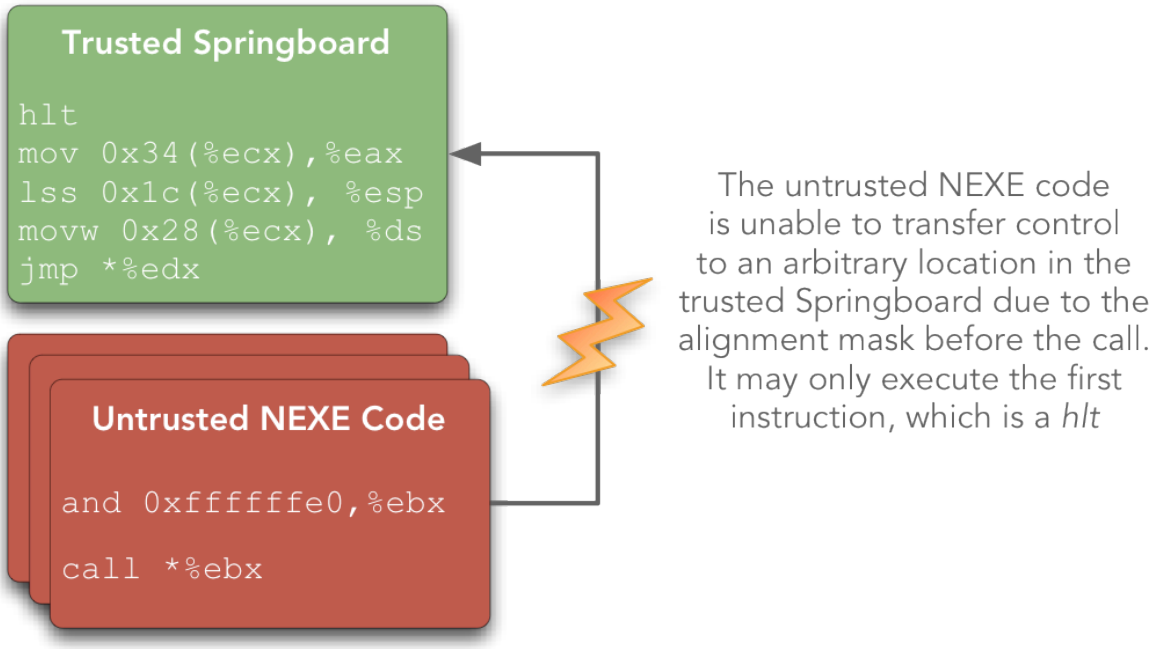
The following is output from the stand alone validator processing the NEXE `hw.nexe` module. It's code has been modified to contain the line `__asm__("ret")` which tells GCC to manually insert a `ret` instruction in that point of the code.

```
$ ./ncval_x86_32 ../examples/hello_world_glibc/hw.nexe
segment[0] p_type 3 p_offset ee0 vaddr 11000ee0 paddr 11000ee0 align 1
    filesz 19 memsz 19 flags 4
segment[1] p_type 1 p_offset 140 vaddr 1000140 paddr 1000140 align 65536
    filesz da0 memsz da0 flags 5
parsing segment 1
VALIDATOR: 10008e6: ret instruction (not allowed)
VALIDATOR: 10008e6: Illegal instruction
segment[2] p_type 1 p_offset ee0 vaddr 11000ee0 paddr 11000ee0 align 65536
    filesz 478 memsz 478 flags 4
segment[3] p_type 1 p_offset 1358 vaddr 11011358 paddr 11011358 align 65536
    filesz 13c memsz 13c flags 6
segment[4] p_type 1 p_offset 10000 vaddr 11020000 paddr 11020000 align 65536
    filesz 0 memsz 2c flags 6
segment[5] p_type 2 p_offset 136c vaddr 1101136c paddr 1101136c align 4
    filesz d8 memsz d8 flags 6
segment[6] p_type 1685382481 p_offset 0 vaddr 0 paddr 0 align 4
    filesz 0 memsz 0 flags 6
segment[7] p_type 7 p_offset 0 vaddr 0 paddr 0 align 4
    filesz 0 memsz 0 flags 4
*** ../examples/hello_world_glibc/hw.nexe IS UNSAFE ***
Validated ../examples/hello_world_glibc/hw.nexe
*** ../examples/hello_world_glibc/hw.nexe IS UNSAFE ***
```

The service runtime, also known as `sel_ldr`, is where the inner sandbox is enforced. The x86 has a little used feature that can be used to segment the memory of a userland process. This is done through the use of segment registers. The segment registers that NaCl utilizes are `%ds (data)`, `%cs (code)` and `%gs (TLS)`. The other remaining segment registers `%es`, `%fs` and `%ss` are all initialized to the value of `%ds`. This is because some instructions use their values implicitly so they must be initialized to constrain access to the untrusted address space.

The service runtime's virtual memory is divided between untrusted and trusted code and data. On Windows all allocations are performed in 64KB chunks due to `VirtualAlloc`, which can not allocate smaller chunks of memory. The first 64KB of untrusted address space is broken up as follows. The first 4KB is protected to catch NULL pointer dereferences from untrusted code. The remaining 60KB is reserved for code that performs transitions between untrusted and trusted code. Untrusted instructions are mapped immediately after. The `%cs` register is used to restrict all execution control transfers within this range. The other segment registers will restrict access to the entire untrusted address space.

Transitioning execution between untrusted and trusted code on the x86 is done through the use of the segment registers, trampolines and springboards. When untrusted code needs to transition to trusted code the `%ds` segment register is reset and a `far call` is made to reset the `%cs` segment register. This is what is known as a trampoline. Once execution in the trusted service runtime has been resumed all segment registers are reset and a context switch is made back to untrusted code. This context switch between trust boundaries is necessary because there should be no shared data between trusted and untrusted threads such as the stack. TLS (Thread Local Storage) is used to restore saved register states. Likewise returning from trusted code to untrusted code a return springboard is mapped into this region. Upon return to the untrusted code the stack pointer is reset and control is transferred back to an arbitrary address in the untrusted NEXE instructions. It is important to note that because the trampolines and springboards execute privileged instructions they are only mapped in by the service runtime during the loading process. Both trampolines and springboards require privileged instructions in order to work. For this reason the first instruction of the springboard code is a `hlt`. This is so that untrusted code can not invoke the trusted code block (execution transfers in untrusted code are only allowed to properly aligned addresses). This is shown in the graphic below:



The trampoline and springboard code can be found in hand written in assembly in Chrome source tree at:

`src/native_client/src/trusted/service_runtime/arch/x86_32/tramp_32.S`
`src/native_client/src/trusted/service_runtime/arch/x86_32/springboard.S`

Trampolines and springboards are essential to NaCl's usability, there is no other way for untrusted NEXE code to make privileged calls and not compromise the security of the service runtime. No direct syscalls may be made by the NEXE module, the validator will refuse to load an executable that contains a `sysenter` instruction. This is required in order to keep the module from performing any privileged calls.

Trampolines and springboards provide an effective mechanism for safe transitions between trusted and untrusted code, however that does not mean a NEXE module can make any syscall it chooses. Allowing the untrusted NEXE module to make arbitrary syscalls would destroy the security of the system. For example, an attacker could easily remove sections of validated code and replace them with new malicious code to escape the inner sandbox. To remedy this the service runtime exposes what are known as `NACL_SYSCALLS`. These are syscalls unique to NaCl that allow untrusted NEXE modules to make simple calls like `open`, `close`, `read`, `write`, `ioctl` and others to create and operate on IMC sockets. These are not designed to be directly called by NEXE developers. Instead they are normally invoked by using the higher level PPAPI library and the built-in IRT (Integrated Runtime) library provided by the SDK.

The NaCl inner sandbox is implemented differently on ARM and x86_64 architectures. This is due to fundamental differences in the instruction sets and their lack of a memory segmentation model similar to that found on the x86. Like the approach taken on the x86 NaCl maintains a list of banned instructions and enforces proper code alignment through its modified toolchain and runtime branch target alignment enforcement. Both the ARM and the x86_64 port, like the 32 bit x86 version, use ILP32 (`int`, `long`, `pointer`) for primitive data types.

On the ARM platform the untrusted address space is allocated below the trusted service runtime. This untrusted address space ranges from `0x00000000` to `0x3fffffff` All loads, stores and branch targets must occur within this pre-allocated space. Guard pages are mapped on either side of the untrusted code to prevent any unbounded buffer over runs from reaching into the trusted address space. Each load or store instruction is preceded by a `tst` instruction that performs a bitwise `AND` operation of the source or destination register value against the constant `0xc0000000`. This is to determine whether the value stored in the register is within the untrusted address space. The `tst` instruction sets the status flags accordingly. The load is performed via the `ldreq` 'load if equal' instruction which will only perform the load if the `Z` status flag is set. An example of the entire sequence is shown below:

```
tst     R4, #0xC0000000
ldreq   R2, [R4]
```

ARM offers additional memory addressing schemes. NaCl allows all of them except for register-indexed mode, which allows two registers to be added to one another. Register indexing is allowed because the largest index value is 4096 bytes, and the untrusted address space is capped by a guard page 8192 bytes in size.

Enforcing the load/store source or destination is enough to contain the untrusted code. All indirect branches must be performed using the `bx` (branch exchange) and `blx` (branch with link exchange) instructions. On ARM the inner sandbox only has to make sure branch targets aren't outside of the untrusted address space and that they don't transfer control to the second instruction in an aligned load/store sequence. The validator can easily check these for direct branches and throw away any modules that don't conform to these rules. Indirect branches pose a tougher problem. Making sure indirect branches don't target an address outside of the untrusted sandbox is done by performing a bitwise `AND` operation similar to that of load/store sequences. Making sure that the indirect branch target isn't the second instruction in a trusted load/store operation (`ldreq`) is enforced similar to that on x86 indirect branches. The indirect branch target first has its top two and bottom four bits cleared which forces the target to be 16 byte aligned. These 16-byte code chunks are referred to as bundles in the NaCl documentation and implementation. No trusted load/store sequences can ever straddle this boundary. These 16 byte aligned code sequences are also used to call into trusted trampolines which make calls into the trusted address space as well as support ARMs unique feature of allowing embedded data in the instruction stream. At the time of this writing the ARM compiler is only available in the PNaCl SDK.

NaCl on x86_64 has its own unique problems and pitfalls. On 32 bit Windows a new Chrome process is launched and the service runtime is statically compiled within it. The Chrome browser is a 32 bit process on 64 bit Windows but the service runtime runs as a stand alone 64 bit process. It is however launched by Chrome and maintains the isolation provided by the Chrome outer sandbox. Like ARM, the x86_64 architecture does not contain the memory segmentation model that the 32 bit x86 does. Because the x86_64 port uses ILP32 data types it limits the address space to 4GB for the service runtime. Like the ARM implementation the untrusted address space is capped by guard regions protecting the trusted address space from unbounded buffer over runs. Access to the `rbp` and `rsp` registers are masked to ensure they always point to addresses in the untrusted region. NaCl hijacks the use of the `r15` register and sets its value to the base of the untrusted code region. All instructions that update the `rip` register must use the `r15` register. The reason for this is so that modifications to `rbp`, `rsp` and `rip` can be based off this value.

This document does not attempt to capture the entire inner sandbox or software fault isolation implementation for ARM and x86_64. We refer you to the official NaCl documentation and code for more information on these topics.

SRPC/IMC

In order for a NEXE module to communicate with the browser a low level interface must be available to transmit the messages back and forth. NaCl provides these interfaces in the form of SRPC which is built on top of the IMC. The service runtime exposes several functions for creating these sockets through the use of `NACL_SYSCALLS`. As noted earlier some of these syscalls are not designed to be invoked directly by a NEXE module. They sit well beneath the PPAPI interfaces NEXE modules normally want to use to communicate with Chrome and the user.

The IMC implementation is built on top of NaCl sockets which are abstractions on top of the OS provided Unix-style sockets, named pipes and shared memory API's. The code for their implementation can be found at the following directory with each OS specific implementation in the `win`, `linux` and `osx` subdirectories:

```
CHROME/src/native_client/src/shared/imc
```

The main API functions for sending and receiving messages are `NaClSendDatagram`, `NaClSendDatagramTo` and `NaClReceiveDatagram`. The message format passed across these sockets is a `NaClMessageHeader` and is defined as follows:

```
/* Message header used by NaClSendDatagram() and NaClReceiveDatagram() */
typedef struct NaClMessageHeader {
  NaClIOVec*  iov;            /* scatter/gather array */
  size_t      iov_length;     /* number of elements in iov */
  NaClHandle* handles;        /* array of handles to be transferred */
  size_t      handle_count;   /* number of handles in handles */
  int         flags;
} NaClMessageHeader;
```

The IMC layer is almost never directly touched by application developers. Instead it is indirectly used by the higher level SRPC layer.

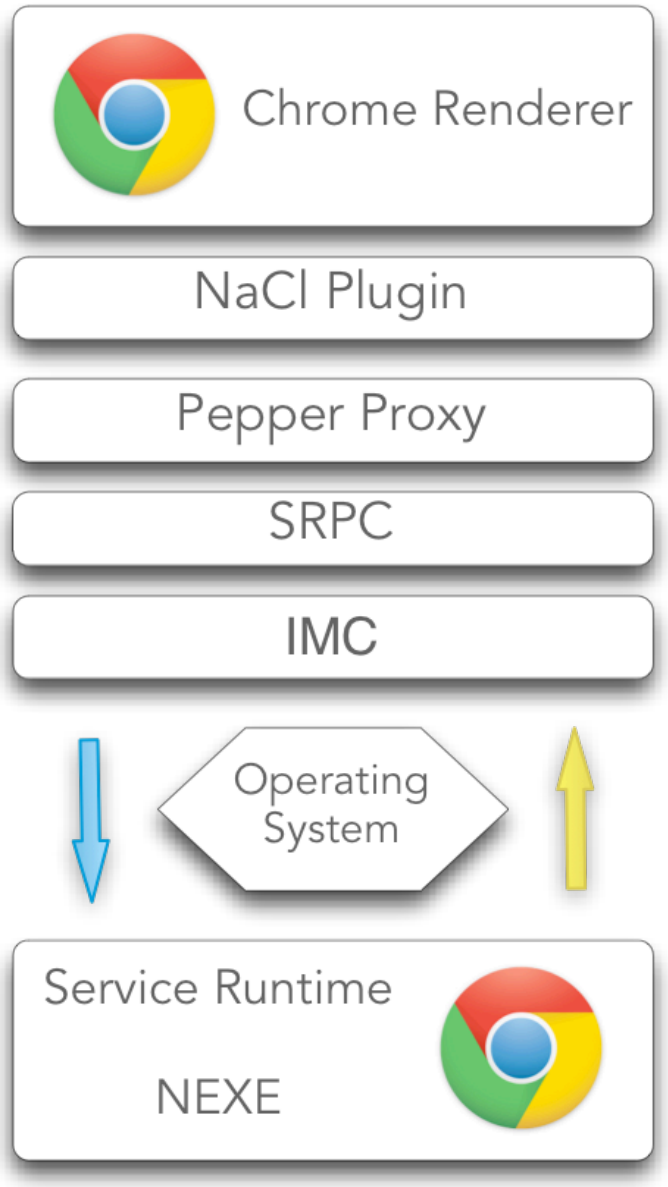
NaCl SRPC is a low level interface for moving messages between untrusted code by way of the the service runtime to Chrome. SRPC is designed to move basic data types between a client and a server. The NEXE side of the SRPC connection exists solely in untrusted code.

SRPC messages are sent and received using the `NaClSrpcMessageChannelSend` and `NaClSrpcMessageChannelReceive` API's. Messages are in the form of `NaClSrpcMessageHeader` structures which is a `typedef` to `NaClMessageHeader` displayed above with their data stored in `NaClSrpcArg` format. The `NaClSrpcArg` format contains a tag value which tells the receiving end what type of data is contained, and two unions. The first union, `u`, holds simple data types such as `bool, int` and `double`. The second union, `arrays`, holds pointers to arrays of these basic types that were serialized in SRPC messages. SRPC servers expose interfaces that can be invoked by the client by using the `NaClSrpcInvokeBySignature` API. An example call is shown below:

```
retval = NaClSrpcInvokeBySignature(
     _channel,
     "MyMethod:i:i", resource, out_bool
 );
```

The example above calls the remote `MyMethod` interface on the channel `_channel`, with two integer arguments `resource` and `out_bool`. Although SRPC is an abstraction layer above IMC it is rarely used by application developers directly.

The following graphic shows where the SRPC and IMC code live in the NaCl communication protocol stack:



PPAPI - NaCl Pepper Proxy

PPAPI (Pepper Plugin Application Programming Interface) is a new plugin API standard proposed by Google to replace the aging NPAPI (Netscape Plugin Application Programming Interface). With PPAPI there are two types of plugins. Trusted plugins, which live in the Chrome browser and untrusted plugins, which exist as NaCl modules. PPAPI also supports out-of-process plugins via the Chrome IPC mechanism. One example of a popular PPAPI plugin is Adobe Flash Player. The Flash Player plugin was converted to PPAPI in early 2012 and runs inside the sandboxed Chrome renderer process.

NaCl modules that use the pepper APIs should use the NaCl SDK directly and not the normal PPAPI available through Chrome. The stable NaCl SDK, at the time of this writing, is Pepper 18. The official SDK contains documentation on each supported interface and it's associated C++ wrapper.

PPAPI introduces a number of new API's for browser plugins including 3D graphics processing, audio, file I/O and more. Their implementation takes care of the low level SRPC details which we explain shortly. Despite its obvious advantages over NPAPI, Google Chrome is currently the only browser to implement PPAPI. PPAPI is completely independent of NaCl. Adobe Flash is one of the first major PPAPI plugins in Chrome.

The PPAPI interfaces live in the sandboxed Chrome renderer process, isolated from the NaCl service runtime and untrusted NEXE modules. In order for NEXE modules to make use of these interfaces a proxy was developed to serialize and deserialize these calls. This is referred to as the 'pepper proxy' in Chrome documentation and source code, it is specific to NaCl and is not used by PPAPI plugins that run in Chrome renderer process. The pepper proxy works over the SRPC interface provided by the NaCl plugin, which in turn uses the IMC interface for low level socket communications. This means that the pepper proxy on the trusted side is responsible for managing objects related to the SRPC communication between itself the remote NEXE.

The pepper proxy is designed to serialize and deserialize function arguments from the untrusted NEXE module. These arguments are either basic types or serialized in the form of a `PP_Var`. The `PP_Var` type is explained below:

`PP_VarType` - An `enum` that specifies the type contained within a `PP_Var`
`PP_VarValue` - A `union` that holds any of the data types specified in `PP_VarType`
`PP_Var` - A structure that holds both a `PP_VarType` and `PP_VarValue`

The `PP_VarValue` structure contains the following fields used to hold data:

`PP_bool as_bool` - Holds a boolean type
`int32_t as_int` - Holds a 32 bit integer
`double as_double` - Holds a double
`int64_t as_id` - Specifies a handle set by the browser. Used if the type is `PP_VARTYPE_STRING`, `PP_VARTYPE_OBJECT`, `PP_VARTYPE_ARRAY`, or `PP_VARTYPE_DICTIONARY`.

A `PP_Var` type can be initialized from a NEXE module using the following C functions:

```
PP_MakeUndefined()
PP_MakeNull()
PP_MakeBool(PP_Bool)
PP_MakeInt32(int32_t)
PP_MakeDouble(double)
```

In the C++ API these are wrapped using the `Var` class which contains similar methods. The `PP_Var` type is commonly used throughout the API for moving data across the pepper proxy in addition to basic types.

There is a fair amount of proxy code shared between both the trusted and untrusted sides. In general it is safe to assume that all interfaces with the prefix `PPP` are implemented on the untrusted side. Conversely all interfaces with the prefix `PPB` are implemented in the browser and are trusted. For example the `PpbURLLoaderRpcServer` class contains the implementations for trusted pepper proxy endpoints for the `URL_Loader` interface. To further confuse those brave enough to read the NaCl source code sometimes interfaces are implemented on both ends of the connection. At the time of this writing the `srpcgen.py` script is used to automatically generate header files for all of these interfaces. Their output is useful in differentiating between trusted and untrusted interfaces. These files can be found at the following paths:

```
src/ppapi/native_client/src/shared/ppapi_proxy/trusted/srpcgen/ppp_rpc.h
Defines RPC client functions in the trusted side

src/ppapi/native_client/src/shared/ppapi_proxy/trusted/srpcgen/ppb_rpc.h
Defines RPC server functions in the trusted side

src/ppapi/native_client/src/shared/ppapi_proxy/untrusted/srpcgen/ppp_rpc.h
Defines RPC server functions in the untrusted module

src/ppapi/native_client/src/shared/ppapi_proxy/untrusted/srpcgen/ppb_rpc.h
Defines RPC client functions in the untrusted module
```

Examining their contents greatly reduces the confusion related to where a particular interface is implemented and what client functions exist to invoke it.

Below is an illustration showing the end-to-end flow of data between an untrusted NEXE module and the trusted pepper API's hosted in the Chrome browser. This example is taken from a NEXE module that utilizes the PPAPI interface `Graphics2D` and calls the `Create` method.


Untrusted NEXE
iface = GetBrowserInterface(PPB_GRAPHICS_2D_INTERFACE)
iface->Create(data)
Springboard
Trampoline
Trusted Service Runtime
NACL_SYSCALL(IMC)(data)
SRPC / IMC
Chrome Renderer Process - NaCl Plugin PPAPI_Proxy
PPB_Graphics2D_CreateDispatcher(data)
PpbGraphics2DRpcServer::PPB_Graphics2D_Create(data)
data = DeserializeTo(data)
(PPAPI Thunk)
PPB_Graphics2D_Impl::Create(data)

Native Client Chrome Browser Plugin

The NaCl PPAPI plugin runs in the sandboxed Chrome renderer process. The plugin has a number of security-sensitive responsibilities. Its primary responsibility is to serve as the binding between trusted components, such as PPAPI and the untrusted code. The NaCl plugin is rather simple in comparison to the service runtime.

It is the responsibility of the NaCl plugin to parse the manifest file covered earlier in this document. The NEXE modules defined in these files may live on the local disk or a remote web site. It is the plugin's responsibility to download or read that file from the disk. The NaCl plugin uses the PPAPI interfaces `URLLoader` and `FileIO` interfaces to request that the browser retrieve and read the NEXE file.

Once the NEXE module has been downloaded and read from disk the service runtime and secure loader must be invoked to load and validate the NEXE. The plugin starts this process by instantiating a class named `ServiceRuntime`. This class abstracts the invoking of the service runtime and is used to manage an SRPC channel to it for sending administrative commands. The NaCl module requests that the Chrome broker process launch the proper service runtime. Please see earlier sections of this document for an explanation on how the appropriate service runtime is selected.

Once the NEXE module is running the NaCl plugin establishes an SRPC channel between the service runtime and itself so that the pepper proxy calls can be made. The pepper proxy itself runs within the NaCl plugin. As covered in the previous section the pepper proxy hosts the SRPC server interfaces and SRPC client code for remote PPAPI calls to and from the untrusted NEXE module.

Because the NaCl module must expose itself to the DOM in order for NEXE modules to be instantiated and communicate with the user, certain bindings are created. These bindings allow for callbacks to be created for exchanging data, responding to certain events and defining properties such as `readyState`, `lastError` and `exitStatus`. These bindings should not be confused with the scripting capability provided by NPAPI, PPAPI offers no such capabilities. When it comes to sending data between JavaScript and NEXE modules this is done through the `postMessage` function of the module's JavaScript object. In order to receive the string sent from JavaScript the NEXE module must implement the `pp::Instance::HandleMessage` function. When the NEXE module wants to send data back to JavaScript this is done through the `PPB_Messaging::PostMessage` function. The JavaScript will be notified of this data if it has registered an event listener for the `message` event. The example below shows how this is implemented in JavaScript.

```
<script>
function handleMessage(e) { document.write(e.data) }
d = document.getElementById('d');
d.addEventListener('message', handleMessage, true);
nexe_module_object = document.getElementById("nexe");
nexe_module_object.postMessage("Hello from JavaScript!");
</script>
...
<div id='d'><embed name="NaCl_module"
           id="nexe"
           width=200 height=200
           src="hello_world.nmf"
           type="application/x-NaCl" /></div>
```

Arbitrary binary and ASCII data can be sent over the `PPB_Messaging` interface in this way. This is the primary way NEXE modules and web pages exchange data as the user interacts with the application. It is up to the developer to define the format of the application data sent over this interface. In other words it will be delivered as it was originally sent by the remote side.

Attack Surface

Attack surface in NaCl is defined any interfaces or code where an attacker may be able to find vulnerabilities that allow untrusted code to access privileged resources. In this section we examine where the NaCl architecture exposes attack surface to untrusted NEXE modules. We will also discuss ways the architecture attempts to mitigate some of these issues. Finally we cover what hurdles an attacker must overcome in order to reach privileged resources outside of the sandbox.

The Chrome web browser ships with a strong sandbox for all supported platforms. It is considered the leading sandbox implementation and has been repurposed by other large software products such as Adobe Reader X. Google has gone to great lengths to ensure vulnerabilities in code such as WebKit can't be used to completely compromise a Chrome user's entire system. It makes sense that NaCl would attempt to capitalize on this existing infrastructure to help mitigate vulnerabilities it may be introducing.

Native Client is an in-process plugin that lives within the Chrome renderer process. All Chrome renderer processes are sandboxed by default on Linux, Windows and OS X. In order to support the software fault isolation security mechanism the untrusted NEXE module lives within the secure loader's address space, which is a separate process from the renderer. The secure loader is executed by the Chrome browser process and resides in the same sandbox as Chrome renderer processes. Due to this design choice a malicious NEXE module would first have to execute code in the context of the secure loader process in order to attempt breaking out of the outer Chrome sandbox.

The common method of escaping the Chrome outer sandbox is through a weakly sandboxed process such as the Flash Player plugin or the GPU process or via a kernel exploit that the sandbox cannot protect against. NaCl's inner sandbox prevents kernel exploits by not allowing the NEXE to interface directly with the kernel. Therefore an attacker must research other avenues for escaping the inner sandbox, even if it only results in outer sandbox privileges. Alternatively an attacker can compromise the remote Chrome renderer process hosting the pepper proxy through a PPAPI endpoint that contains a vulnerability. We explore these routes through various code paths in the text below.

Before we move on, the question of "What about vulnerable NEXE modules" is asked enough that we feel it should be covered on its own. Vulnerabilities in NEXE modules themselves are irrelevant and considered a non-issue. This is mainly due to the same reasons an untrusted NEXE module should not be able to execute code that has not been validated. All dynamic calls are checked to ensure their location is properly aligned, with validated code syscalls that would enable the mapping of new executable memory are thoroughly validated and trusted springboards and trampolines begin with a `hlt` instruction because their alignment allows for untrusted code to invoke them directly.

Service Runtime Inner Sandbox

The ELF file format is a binary file format that can be complex to parse. There have been many vulnerabilities reported in the past related to kernel ELF loading code with integer overflow related issues. Trusted code in the service runtime is responsible for parsing the ELF structures found in the NEXE module. This is a critical phase of the loading process and is the first place a malicious NEXE file may attack privileged code. These potential weak points include parsing of the ELF header and Program headers.

At the time of this writing no vulnerabilities have been publicly reported in this code. NaCl originally shipped with a fuzzer designed by Google specifically for discovering vulnerabilities in the ELF loader. Successfully exploiting a vulnerability in this code would allow an untrusted NEXE module to execute arbitrary code in the context of the service runtime, which of course is sandboxed in a Chrome renderer process.

The inner-sandbox and code validation routines is another area an attacker may find and exploit vulnerabilities. A successful attack on the inner sandbox may be discovering a way to bypass syscall filtering and map in new executable code that hasn't been validated. Defeating the code validator may come in the form of using an obscure instruction or relying on the side effects of a valid instruction that the NaCl authors had not anticipated. This could include prefix bytes in branch instructions or instructions considered benign and allowed by the validator. Discovering a parsing flaw in the code validator itself would also allow for an attacker to execute arbitrary code in the context of the sandboxed service runtime.

The service runtime is also responsible for handling all NACL_SYSCALL requests. While there are less than 40 of these defined, a vulnerability in their implementation would allow an untrusted NEXE module to execute code in the context of the trusted service runtime. Luckily the amount of code exposed here is minimal and can be fuzzed and audited thoroughly.

PPAPI - NaCl Pepper Proxy

It is arguably easier for an attacker to escape the untrusted NEXE module by discovering and exploiting a vulnerability in a remote PPAPI interface via the pepper proxy. This route avoids the inner sandbox all together. The number of API end points available through the PPAPI interface that parse untrusted data is large and growing with each release. The proxy itself is not a security mechanism and does not always validate data as well formed as this is not always possible. In addition to parsing untrusted data these interfaces are responsible for tracking related SRPC channels and callback triggers related to their functionality.

Discovering and exploiting a vulnerability in a remote PPAPI interface would allow for executing arbitrary code in the context of the Chrome renderer process. As stated earlier, this constitutes a defeat of the NaCl security model.

These remote interfaces sometimes pass data off to other components in Chrome such as the GPU process. In cases like this an attacker may be able to completely escape the Chrome sandbox or execute code within a less restrictive version of it. This was the case with PinkyPie's Chrome exploit at the 2012 Pwnium competition at the CanSecWest conference. This impressive chain of exploits partially used NaCl as a vehicle to escape the Chrome sandbox. In the initial stages of the exploit PinkyPie was able to trick Chrome into loading a NEXE module from an arbitrary location. The second link in the exploit chain was to send data via that NEXE module to a remote GPU API interface that triggered an integer overflow in the Chrome GPU process. Exploiting this vulnerability via a NEXE module was possible due to the raw access NaCl provides to the GPU command buffers. This would have been far more difficult to perform if it required exploiting a WebKit vulnerability first and then executing shellcode in the Chrome sandbox to send the same data to the GPU process.

In a later section of this document we cover a number of vulnerabilities found in the pepper proxy by the author of this document.

SRPC/IMC

The service runtime hosts a small amount of trusted IMC code. This interface can be reached via specific `NACL_SYSCALLS` by untrusted code. This code is responsible for setting up the sockets and shared memory used to support the IMC protocol. Discovering and exploiting a vulnerability in these functions would result in running arbitrary code within the context of the sandboxed service runtime.

However the NaCl plugin hosted in the renderer contains a large amount of trusted SRPC and IMC code in order to support the pepper proxy. As discussed in previous sections the SRPC protocol can be a complex binary stream of serialized data. These low level receivers and dispatchers are a valid target for a malicious NEXE to exploit. Any arbitrary code executed here will run with the privileges of the Chrome renderer process.

NaCl Browser Plugin

The NaCl browser plugin creates additional attack surface that can be reached via untrusted web content. This includes the parsing of untrusted JSON NEXE manifest files and the JavaScript bindings created when the plugin is instantiated. Even though JSON is a simple format to parse this is a particularly security sensitive area. NaCl currently uses a third party library, jsoncpp, to parse the JSON. Although this library has been well tested it still may contain vulnerabilities. The DOM bindings for NaCl modules are relatively slim compared to other browser plugins. They exist to provide the web application the ability to send and receive data from the NEXE, respond to certain events and retrieve status codes. Regardless of how slim this scripting capability is, it binds untrusted JavaScript to C++ code in the browser.

Vulnerabilities Identified

A number of vulnerabilities have been reported in NaCl since it was first announced in 2009. A majority of these vulnerabilities allow for the execution of arbitrary code in trusted components and a select few bypass the inner sandbox by directly executing code that had not been validated.

In this section we examine the first vulnerability discovered in NaCl, some of the vulnerabilities found during the 2009 security contest, vulnerabilities discovered by Google developers and more recently during a source code review the author performed for Google in June 2011.

First Independently Discovered NaCl Vulnerability

The very first vulnerability reported in NaCl by an outside party was an inner sandbox break out discovered by security researcher named Alex Radocea in 2008.

Call Instruction Memory Dereference (link)

The NaCl instruction validator failed to take memory dereferences into account for `call` instructions. This allows the following instruction to transfer execution to an unvalidated location because the validator has only ensured the register value is 32 byte aligned but not the value it references.

```
andl $0xffffffe0, %edx
call *(%edx)
```

2009 Security Contest

Google held a security contest in 2009 that invited security researchers to find vulnerabilities in both the implementation and the design of NaCl. This included the NPAPI plugin, IMC/SRPC and the inner sandbox.

The contest resulted in a number of implementation related vulnerabilities with a total of 20 unique vulnerabilities reported. Despite the fact no entries irreparably broke the inner sandbox design, a number of these issues were either unique to NaCl's security boundaries or allowed for sandbox breakouts via arbitrary code execution in trusted components. The architecture of NaCl has significantly changed since this contest was held so we only briefly review some of the more interesting findings from that contest and how they relate to the current code base.

Unchecked 2-byte jmp Instruction Prefix (link)

This vulnerability allowed attackers to transfer execution to code that had not been properly validated. This is a good example of why validating complex instruction sets like the x86 is difficult. This vulnerability resulted in an inner sandbox breakout.

EFLAGS Direction Flag Modification (link)

The x86 EFLAGS status register contains a status flag that represents the direction data is to be copied in memory. NaCl allowed untrusted modules to set this flag which could result in memory corruption and arbitrary code execution once the context switch to trusted code was made. This vulnerability shows that side effects of the architecture that are controlled via non-privileged instructions may result in serious vulnerabilities.

Validated Code Unmapping (link)

After validation an untrusted NEXE module could unmap validated code and remap it with unvalidated code to escape the inner sandbox. While this vulnerability was easily patched by properly validating the syscall arguments it is another example of how the inner sandbox can be defeated by allowing the execution of unvalidated code.

Uninitialized vtable (link)

An integer overflow, an unchecked return value and an uninitialized heap buffer lead to potential arbitrary code execution in trusted components. This vulnerability was triggered via untrusted JavaScript. Some of the NaCl C code is written as object oriented with manual constructor, destructor and vtable setup to handle callbacks and function pointers. This requires carefully initializing and managing structures that would normally be managed automatically in C++.

Double delete Operator (link)

The NaCl plugin relied on the NPAPI interfaces to manage the lifetimes of certain browser binding related objects. A specific HTML sequence that triggered an error condition would cause the plugin to delete an object already deleted by the NPAPI backend. While NPAPI is no longer used the same issues may be found in the PPAPI bindings where there is confusion over who is responsible for the lifetime of an object. This is true of any PPAPI plugin and is not unique to NaCl.

Public NaCl Vulnerabilities

A small number of vulnerabilities have been reported by Google developers and are publicly available on the projects website. The NaCl issue tracker contains a number of vulnerabilities and security relevant bugs however only issues reported within 2011 and 2012 are listed below.

Inner Sandbox Escape via KiUserExceptionDispatcher (link)

On 64 bit Windows 7 systems when a fault occurs the kernel transfers execution to the `KiUserExceptionDispatcher` function in ntdll.dll. When this function is invoked it walks a list of vectored exception handlers on the stack. However this function does not know the difference between the trusted and untrusted NaCl stacks. The `KiUserExceptionDispatcher` function will call other functions that end in a `ret` instruction. When this occurs if the `%rsp` register points at the untrusted stack this will redirect execution to a location of the attackers choosing. This is exploited by writing to the first threads stack using a second thread that will continue executing while `KiUserExceptionDispatcher` is busy.

BSF Instruction Inner Sandbox Escape (link)

A 64 bit NEXE can escape the inner sandbox by using the `bsf/bsr` instruction. The `bsf` instruction can be used to write to an arbitrary location in the process memory. This is possible because the `bsf` instruction only conditionally writes to its destination register if its source register is zero. A subsequent `mov` operation can not guarantee the `BSF` instruction properly masked the destination register.

Trampoline Trusted Address Space Leak (link)

The trusted trampoline code in 64 bit NEXE modules directly transfers control to the trusted syscall handler. An untrusted module can read this code and determine where the .text is mapped in the trusted service runtime. This can be used to defeat ASLR when exploiting a code execution vulnerability in the service runtime.

Address Space Leak From JavaScript Error Messages (link)

In certain cases a detailed error string would be generated by the trusted plugin and retrievable via JavaScript. This error string contained valid memory addresses in the renderer process. This can be used to defeat ASLR when exploiting a code execution vulnerability in the service runtime.

2011 Pepper Proxy Code Review

As noted in the architecture section of this document the proxy is quite large and is responsible for serializing and deserializing untrusted binary data to and from both trusted and untrusted components. A total of 10 unique security vulnerabilities were identified during the 3 week audit. Some of these vulnerabilities are found directly in the pepper proxy itself while others are the result of the pepper proxy allowing untrusted and unvalidated data to propagate into the PPAPI interface implementation or other areas of the Chrome renderer process.

PPB_Graphics2D_Create Shared Memory Integer Overflow (link)

The `PPB_Graphics2D_Create` interface receives a serialized `PP_Size` structure which contains signed integers `width` and `height`. These two integers are passed to `PPB_ImageData_Impl::Init` in the PPAPI implementation. There is an attempt to detect overflow in this function:

```
if (static_cast<int64>(width) * static_cast<int64>(height) >=
      std::numeric_limits<int32>::max())
   return false;  // Prevent overflow of signed 32-bit ints.
```

However further down the code path in `PepperPluginDelegateImpl::CreateImage2D` these values are multiplied by each other and 4 which allows for an integer overflow condition to occur:

```
PepperPluginDelegateImpl::CreateImage2D(int width, int height) {
```

```
uint32 buffer_size = width * height * 4;
```

This overflowed value is passed to `TransportDIB::Create` where a shared memory buffer is allocated with the overflowed size:

```
const int shmkey = shmget(IPC_PRIVATE, size, 0666);
```

This memory buffer is shared with the trusted browser broker process.

PPB_Context3DTrusted_CreateTransferBuffer Shared Memory Integer Overflow (link)

The `PPB_Context3DTrusted_CreateTransferBuffer` interface takes a size parameter provided by an untrusted NEXE module. This size parameter is passed to the `CommandBufferService::CreateTransferBuffer` function where it is passed to `CreateAnonymous` and `CreateNamed` functions. On Win32 platforms if `MAX_UINT` is provided for the size it will overflow in `SharedMemory::CreateNamed` in the following line

```
uint32 rounded_size = (size + 0xffff) & ~0xffff;
```

This memory buffer is shared with the trusted browser broker process.

PPB_Audio_Create SRPC Channel Use After Free (link)

When the audio interface `PpbAudioRpcServer::PPB_Audio_Create` is first invoked a new `StreamCreatedCallbackData` instance is created in the renderer process. This callback is created to alert the client when the PPAPI Audio implementation has finished processing the request. If the NEXE module tears down the SRPC channel before the audio stream has been created then a use after free condition will occur when the callback executes and accesses the stale `channel` pointer. This occurs when the callback invokes `PppAudioRpcClient::PPP_Audio_StreamCreated` which calls `NaClSrpcInvokeBySignature` which then dereferences the stale SRPC `channel` pointer in the following call: `NaClSrpcServiceMethodIndex(channel->client, rpc_signature)`

PPB_URLLoader_Open CORS Request Allows For Header Injection (link)

The `PPB_URLLoader_Open` and `PPB_URLRequestInfo_SetProperty` interfaces allow a NEXE module to configure various parameters for an HTTP request that the browser will make on its behalf. In order to perform cross domain requests Webkit is instructed by the PPAPI layer to use the `CrossOriginRequestPolicyUseAccessControl` option. This means the browser will send a 'pre-flight' HTTP OPTIONS request to the third party server asking it what HTTP parameters may be set.

However a NEXE module can use the `SetProperty` call to inject arbitrary HTTP headers into this request by injecting raw CRLF characters into the headers. An example of this is below:

```
set_method = request_interface_->SetProperty(request_,
        PP_URLREQUESTPROPERTY_METHOD,
        Module::StrToVar("POST\x0d\x0ax-csrf-token:\x20test1234"));
```

PPB_URLLoader_ReadResponseBody Heap Overflow (link)

The `PPB_URLLoader_ReadResponseBody` function contains a heap overflow. The following code can be found in the function body for the `URLLoader` SRPC server `ReadResponseBody` function.

```
  *pp_error_or_bytes = PPBURLLoaderInterface()->ReadResponseBody(
      loader, callback_buffer, bytes_to_read, remote_callback);
  DebugPrintf("PPB_URLLoader::ReadResponseBody: pp_error_or_bytes=%"
              NACL_PRId32"\n", *pp_error_or_bytes);

  if (*pp_error_or_bytes > 0) {  // Bytes read into |callback_buffer|.
    // No callback scheduled.
    *buffer_size = static_cast<nacl_abi_size_t>(*pp_error_or_bytes);
    memcpy(buffer, callback_buffer, *buffer_size);
```

`bytes_to_read` is controlled by the untrusted NEXE module. In this case `sizeof(buffer)` should be equal to `buffer_size` since it was allocated by the trusted browser side. The call to `memcpy` will copy `buffer_size` bytes from `callback_buffer` into `buffer` resulting in a heap overflow.

PPB_FileIO_Write Out Of Bounds Read Information Leak (link)

The function `PPB_FileIO_Write` receives `buffer`, and a `bytes_to_read` parameter provided by the untrusted NEXE module. There is no check to ensure `(bytes_to_read < sizeof(buffer))`. An untrusted NEXE module can send a `bytes_to_read` value of `sizeof(buffer)+N` and the `RelayWrite` class will perform an out of bounds read on the buffer. This will result in an information leak where an untrusted NEXE module can retrieve memory contents of the Chrome renderer process hosting the NaCl plugin it is communicating with.

PPB_FileIO_Dev_Read Heap Overflow (link)

The `PPB_FileIO_Dev_Read` function contains a heap overflow. The following code can be found in the function body for the `FileIO` RPC server `Read` function.

```
*pp_error_or_bytes = PPBFileIOInterface()->Read(file_io,
    offset, callback_buffer, bytes_to_read, remote_callback);
DebugPrintf("PPB_FileIO_Dev::Read: pp_error_or_bytes=%"NACL_PRId32"\n",
            *pp_error_or_bytes);

if (*pp_error_or_bytes > 0) {  // Bytes read into |callback_buffer|.
  // No callback scheduled.
  *buffer_size = static_cast<nacl_abi_size_t>(*pp_error_or_bytes);
  memcpy(buffer, callback_buffer, *buffer_size);
```

`bytes_to_read` is controlled by the untrusted NEXE module. In this case `sizeof(buffer)` should be equal to `buffer_size` since it was allocated by the trusted browser side. The call to `memcpy` will copy `buffer_size` bytes from `callback_buffer` into `buffer` resulting in a heap overflow.

PPB_PDF_SearchString Potential Heap Overflow (link)

The `PPB_PDF_SearchString` interface contains an integer overflow that may lead to heap memory corruption. This code is found in a private interface and may require a special compromised plugin in order to reach.

```
int pp_result_count = 0;
PPBPDFInterface()->SearchString(instance,
    reinterpret_cast<unsigned short*>(string),
    reinterpret_cast<unsigned short*>(term),
    case_sensitive ? PP_TRUE : PP_FALSE,
    &pp_results,
    &pp_result_count);
*results_size = std::min(*results_size, pp_result_count *
                            kPpbPrivateFindResultBytes);
memcpy(results, pp_results, *results_size);
free(pp_results);
*count = static_cast<int32_t>(pp_result_count);
```

The `pp_result_count * kPpbPrivateFindResultBytes` multiplication may overflow if `pp_result_count = (MAX_INT/8)+1` which would assign a negative value to `results_size` (which is of type `nacl_abi_size_t`) which would overflow the `results` buffer in the call to `memcpy`.

PPB_FileRef_Create Potential Directory Traversal (link)

The call to `PPB_FileRef_Create` eventually invokes a validation function to `isValidLocalPath` in order to validate the path passed to the interface by an untrusted NEXE module.

```
bool IsValidLocalPath(const std::string& path) {
 // The path must start with '/'
 if (path.empty() || path[0] != '/')
   return false;

 // The path must contain valid UTF-8 characters.
 if (!IsStringUTF8(path))
   return false;

 return true;
}
```

This check is not enough to find directory traversal attacks such as `../../../priv/dir`. The Chrome sandbox will prevent the untrusted NEXE module from overwriting sensitive files outside of the browser but any untrusted paths should be constrained to a specific location for NEXE modules or otherwise filtered for these types of attacks.

Heap Overflow In MessageChannelEnumerate (link)

The function `MessageChannelEnumerate` contains a potential heap overflow. The multiplication performed on the size parameter passed to the `malloc` call based on the number of message channels can potentially wrap and lead to a heap overflow if too small of a buffer is allocated.

Chrome Shaker

In January of 2012 we developed a NEXE module designed to fuzz the pepper proxy. To continue Chrome's *salt* and *pepper* theme we named this fuzzer *chrome-shaker*.

The Chrome browser source code ships with a number of IDL files that describe the various PPAPI interfaces and their supported arguments. The NaCl team relies heavily on these to help auto generate C++ interface code. Below is an example of one such interface description for the `PPB_Messaging` interface:

```
interface PPB_Messaging {
  void PostMessage([in] PP_Instance instance, [in] PP_Var message);
};
```

In order to properly parse these IDL files the NaCl authors have developed a python library that creates an AST (Abstract Syntax Tree). Using this python library we auto generate C++ code that fuzzes the various PPAPI interfaces with random data.

Chrome Shaker is a rather simple approach to fuzzing the pepper proxy. The fuzzer consists of a skeleton NEXE module written in C++ that contains various fuzzing and logging functions, glue code for instantiating the NaCl runtime and HTML scripts that provide a simple user interface. The python code generator takes this skeleton plugin and creates the code for fuzzing each interface described in the IDL files provided in the Chrome source tree.

A number of complications were encountered while trying to create a pepper proxy fuzzer. This is mainly due to requiring PPAPI resources that are only available through the interfaces the NEXE is designed to fuzz.

The first problem we encountered was obtaining a source of random data. A NEXE module can not simply open `/dev/urandom` and read an arbitrary number of bytes. This is strictly prohibited by the sandbox. We developed a work around for this that involved calling a JavaScript function `window.crypto.getRandomValues` and sending a buffer of 2048 random bytes through the `PostMessage` interface to the NEXE module. This buffer of random data is repeatedly refilled during the fuzzers runtime.

Logging is also complicated from a NEXE module under test. This is because in order to write to an external file you must use the `FileIO` interfaces provided by PPAPI. This involves writing to interfaces that are currently being fuzzed. This results in mutex deadlocks and other issues. The current implementation will log all data to `STDOUT` by default and prefix each entry with the string `Shaker`.

Conclusion

The NaCl architecture is indeed large and complex. While the original 'Trusted Code Base' has slowly grown in size to support more functionality, only a small number of bugs have have affected the inner sandbox since its release. NaCl has innovated strong sandbox mechanisms including its software fault isolation and inner sandbox on the x86, x86_64 and ARM platforms. It is very likely that future sandbox approaches will borrow ideas, concepts and code directly from NaCl.

NaCl creates the opportunity to take often used native code components and place them inside two sandboxes. This further raises the cost for an attacker to successfully exploit a vulnerability in code protected by NaCl. One can envision a potential future where Chrome's default PDF reader is simply a NEXE bytecode that is securely compiled on-the-fly to memory in order to parse and render potentially malicious PDF documents from the internet.

The 2009 security contest provided an accurate look at the future of vulnerabilities in NaCl. The contest showed that it was more likely for future vulnerabilities to be found in trusted components that were the result of handling untrusted data than would be found in the inner sandbox or the general software fault isolation design. This has largely held true to the present day with the exception of the x86_64 inner sandbox. The inner sandbox design on the x86 remains strong due to its simplicity and memory segmentation model support. However the x86_64 and ARM inner sandbox designs have yet to be as thoroughly tested as the former.

The future of NaCl is PNaCl or 'Portable Native Client'. PNaCl uses the LLVM AOT (Ahead Of Time) compiler to safely generate native code that conforms to NaCl's inner sandbox rules. This of course will make NEXE modules platform independent, enabling the same application to run on an Android device, Chrome OS or Windows system. PNaCl brings with it a whole new set of architectural and implementation security issues to be considered with. We look forward to researching PNaCl as it comes online.

For the truly paranoid among us, all plugins, including Native Client, should be blocked and explicitly enabled via Chromes click-to-play mechanism.

The author would like to thank Justin Schuh and David Sehr from Google, and Cory Scott and Dave Goldsmith from Matasano for providing the opportunity to work on such an interesting project and for allowing this research to be published at BlackHat.

References

Native Client

https://developers.google.com/native-client/

Chrome Sandbox Design Documents

http://www.chromium.org/developers/design-documents/sandbox/

A Sandbox for Portable, Untrusted x86 Native Code

http://src.chromium.org/viewvc/native_client/data/docs_tarball/nacl/googleclient/native_client/documentation/nacl_paper.pdf

Native Client 2009 Security Contest

https://developers.google.com/native-client/community/security-contest/

Exploiting Native Client - Ben Hawkes (HAR 2009)

http://www.stanford.edu/class/cs240/readings/hawkes_HAR_2009_exploiting_native_client.pdf

Chrome Shaker Project

https://code.google.com/p/chrome-shaker/

Chris Rohlf
Leaf SR
http://leafsr.com
LeafSRInfo@gmail.com
(732) 737-7513