Syscall Proxying - Simulating remote execution

Maximiliano Caceres
<maximiliano.caceres@corest.com>

Table of Contents

Abstract

A critical stage in a typical penetration test is the "Privilege Escalation" phase. An auditor faces this stage when access to an intermediate host or application in the target system is gained, by means of a previous successful attack. Access to this intermediate target allows for staging more effective attacks against the system by taking advantage of existing webs of trust and a more privileged position in the target system's network. This "attacker profile" switch is referred to as *pivoting* along this document.

Pivoting on a compromised host can often be an onerous task, sometimes involving porting tools or exploits to a different platform and deploying them. This includes installing required libraries and packages and sometimes even a C compiler in the target system!.

Syscall Proxying is a technique aimed at simplifying the privilege escalation phase. By providing a direct interface into the target's operating system it allows attack code and tools to be automagically in control of remote resources. Syscall Proxying transparently "proxies" a process' system calls to a remote server, effectively simulating remote execution.

Basic knowledge of exploit coding techniques as well as assembler programming is required.

General concepts

A typical software process interacts, at some point in its execution, with certain resources: a file in disk, the screen, a networking card, a printer, etc. Processes can access these resources through system calls (*syscalls* for short). These *syscalls* are operating system services, usually identified with the lowest layer of communication between a user mode process and the OS kernel.

The `strace` tool available in Linux systems [1] "...intercepts and records the system calls which are called by a process...". From `man strace`:

> Students, hackers and the overly-curious will find that a great deal can be learned about a system and its system calls by tracing even ordinary programs. And programmers will find that since system calls and signals are events that happen at the user/kernel interface, a close examination of this boundary is very useful for bug isolation, sanity checking and attempting to capture race conditions.

For example, this is an excerpt from running `strace uname -a` in a Linux system (some lines were removed for clarity):

```
[max@m21 max]$ strace uname -a
execve("/bin/uname", ["uname", "-a"], [/* 24 vars */]) = 0 ❶
(...)
uname({sys="Linux", node="m21.corelabs.core-sdi.com", ...}) = 0 ❷
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x4018e000
write(1, "Linux m21.corelabs.core-sdi.com "..., 85) = 85 ❸
munmap(0x4018e000, 4096)                = 0
_exit(0)                                = ? ❹

Linux m21.corelabs.core-sdi.com 2.4.7-10 #1 Thu Sep 6 17:27:27 EDT 2001 i686 unknown
[max@m21 max]$
```

❶ The `strace` process forks and calls the `execve` *syscall* to execute the `uname` program.

[1] Similar tools are `ktrace` in BSD systems and `truss` in Solaris. There's a strace like tool for Windows NT, available at http://razor.bindview.com/tools/desc/strace_readme.html.

❷ The `uname` program calls the `uname` *syscall* of the same name (see `man 2 uname`) to obtain system information. The information is returned inside a struct provided by the caller.

❸ The `write` *syscall* is called to print the returned information to file descriptor 1, `stdout`.

❹ `exit` signals the kernel that the process finished with status 0.

Different operating systems implement *syscall* services differently, sometimes depending on the processor's architecture. Looking at a reasonable set of "mainstream" operating systems and architectures (i386 Linux, i386 *BSD, i386/SPARC Solaris and i386 Windows NT/2000) two very distinct groups arise: UNIX and Windows.

The UNIX way

UNIX systems use a generic and homogeneous mechanism for calling system services, usually in the form of a "trap", a "software interrupt" or a "far call". *syscalls* are classified by number and arguments are passed either through the stack, registers or a mix of both. The number of system services is usually kept to a minimum (about 270 *syscalls* can be seen in OpenBSD 2.9), as more complex functionality is provided on higher user-level functions in the `libc` library. Usually there's a direct mapping between *syscalls* and the related `libc` functions.

The Windows way

> The native API is little mentioned in the Windows NT documentation, so there is no well-established convention for referring to it. "native API" and "native system services" are both appropriate names, the word "native" serving to distinguish the API from the Win32 API, which is the interface to the operating system used by most Windows applications.
>
> It is commonly assumed that the native API is not documented in its entirety by Microsoft because they wish to maintain the flexibility to modify the interface in new versions of the operating system without being bound to ensure backwards compatibility. To document the complete interface would be tantamount to making an open-ended commitment to support what may come to be perceived as obsolete functionality. Indeed, perhaps to demonstrate that this interface should not be used by production applications, Windows 2000 has removed some of the native API routines that were present in Windows NT 4.0 and has changed some data structures in ways that are incompatible with the expectations of Windows NT 4.0 programs.
>
> (...)
>
> The routines that comprise the native API can usually be recognized by their names, which come in two forms: `NtXxx` and `ZwXxx`. For user mode programs, the two forms of the name are equivalent and are just two different symbols for the same entry point in `ntdll.dll`.
>
> —Gary Nebbett, *Windows NT/2000 Native API Reference*

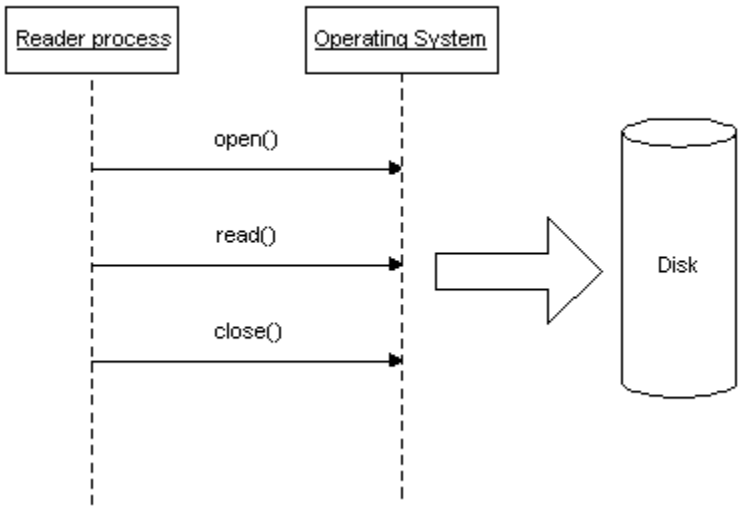
Due to this "lack of definition" for its system services, the high number of functions in this category (more than 1000), and the fact that the bulk of functionality for some of them is part of large user mode dynamic libraries, we'll refer to "Windows *syscalls*" to any function in any dynamic library available to a user mode process. For simplicity's sake, this definition includes higher level functions than those defined in `ntdll.dll`, and sometimes very far above the user / kernel limit.

Syscall Proxying

From the point of view of a process the resources it has access to, and the kind of access it has on them, defines the "context" on which it is executed.

For example, a process that reads data from a file might do so using the `open`, `read` and `close` *syscalls* (see Figure 1).

Figure 1. A process reading data from a file



Syscall proxying inserts two additional layers between the process and the underlying operating system. These layers are the *syscall stub* or *syscall client* layer and the *syscall server* layer.

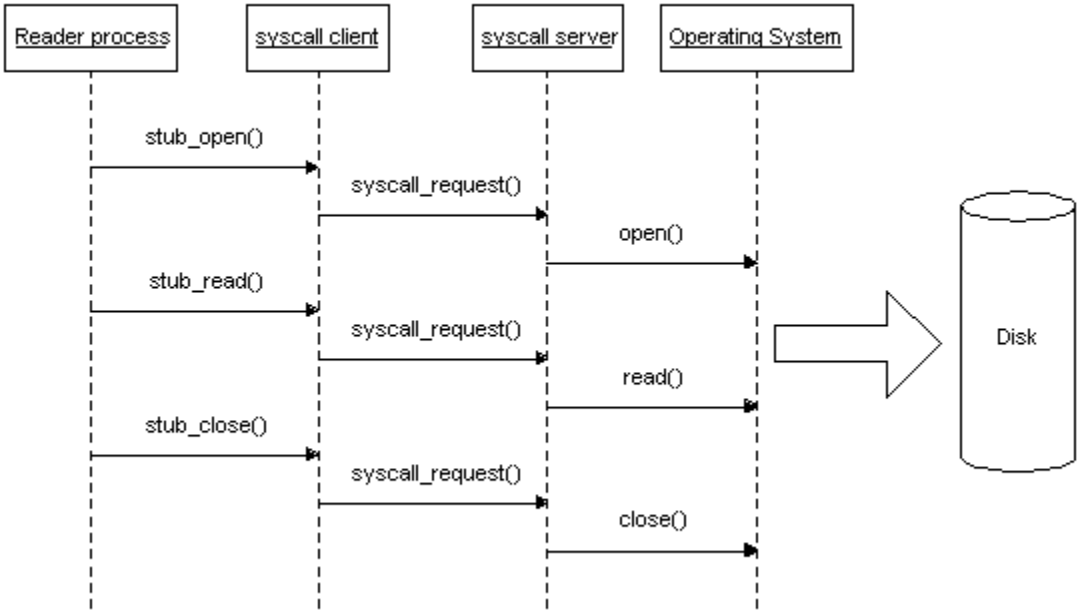
The *syscall client* layer acts as the nexus between the running process and the underlying system services. This layer is responsible for marshaling each *syscall* arguments and generating a proper request that the *syscall server* can

understand. It is also responsible for sending this request to the *syscall server* and returning back the results to the calling process.

The *syscall server* layer receives requests from the *syscall client* to execute specific *syscalls* using the underlying operating system services. This layer marshals back the *syscall* arguments from the request in a way that the underlying OS can understand and calls the specific service. After the *syscall* finishes, its results are marshaled and sent back to the client.

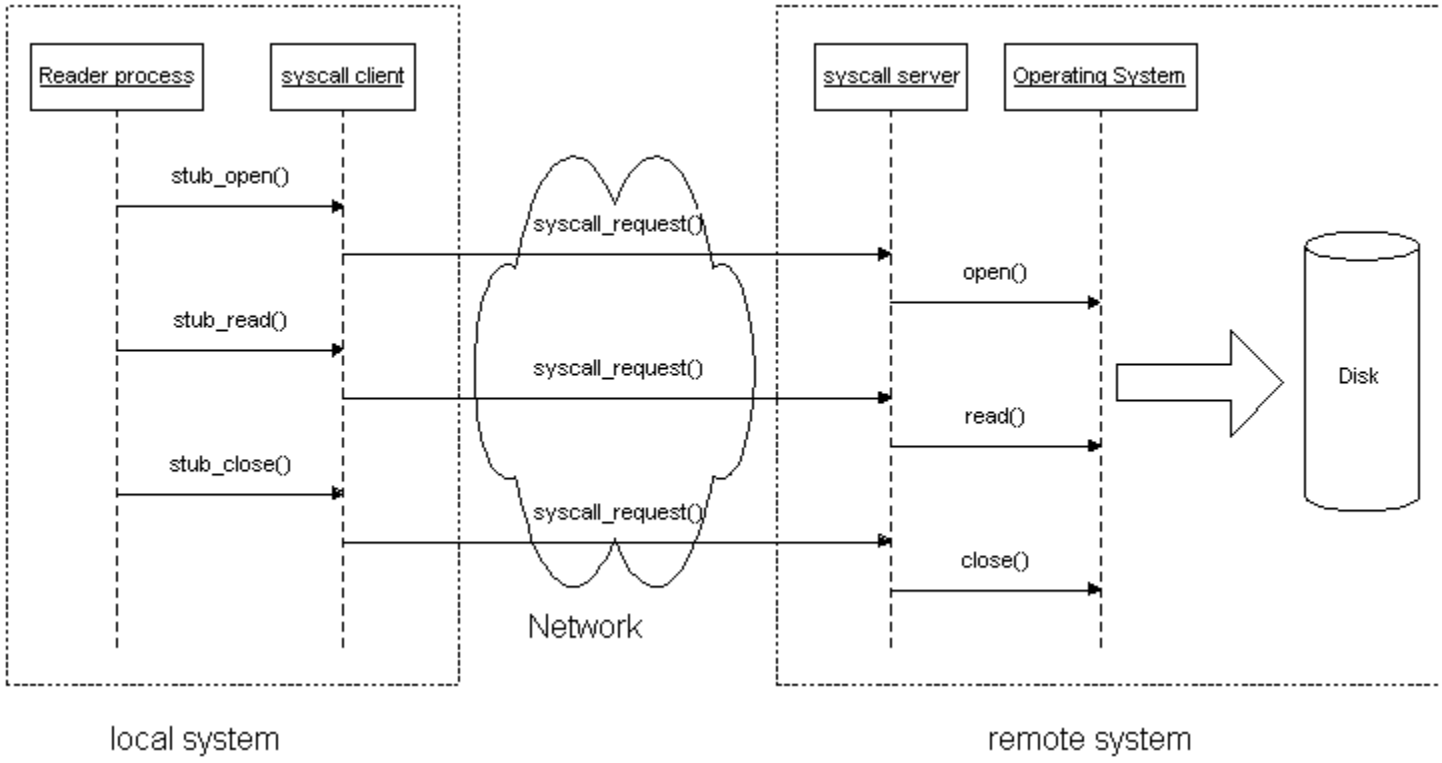
For example, the same reader process from Figure 1 with these two additional layers would look like Figure 2.

Figure 2. A process reading data from a file with the new layers



Now, if these two layers, the *syscall client* layer and the *syscall server* layer, are separated by a network link, the process would be reading from a file on a remote system, and it will never know the difference (see Figure 3).

Figure 3. Syscall Proxying in action



Separating the *syscall client* from the *syscall server* effectively changes the "context" on which the process is executing. In fact, the process "seems" to be executing "remotely" at the host where the *syscall server* is running (and we'll see later that it executes with the privileges of the remote process running the *syscall server*). It is important to note that neither the original file reader program was modified (besides changing the calls to the operating system's *syscalls* to calls into the *syscall client*) to accomplish remote execution, nor did its inner logic change (for example, if the program counted occurrences of the word 'coconut' in the file, it will still count the same way when working with the remote server).

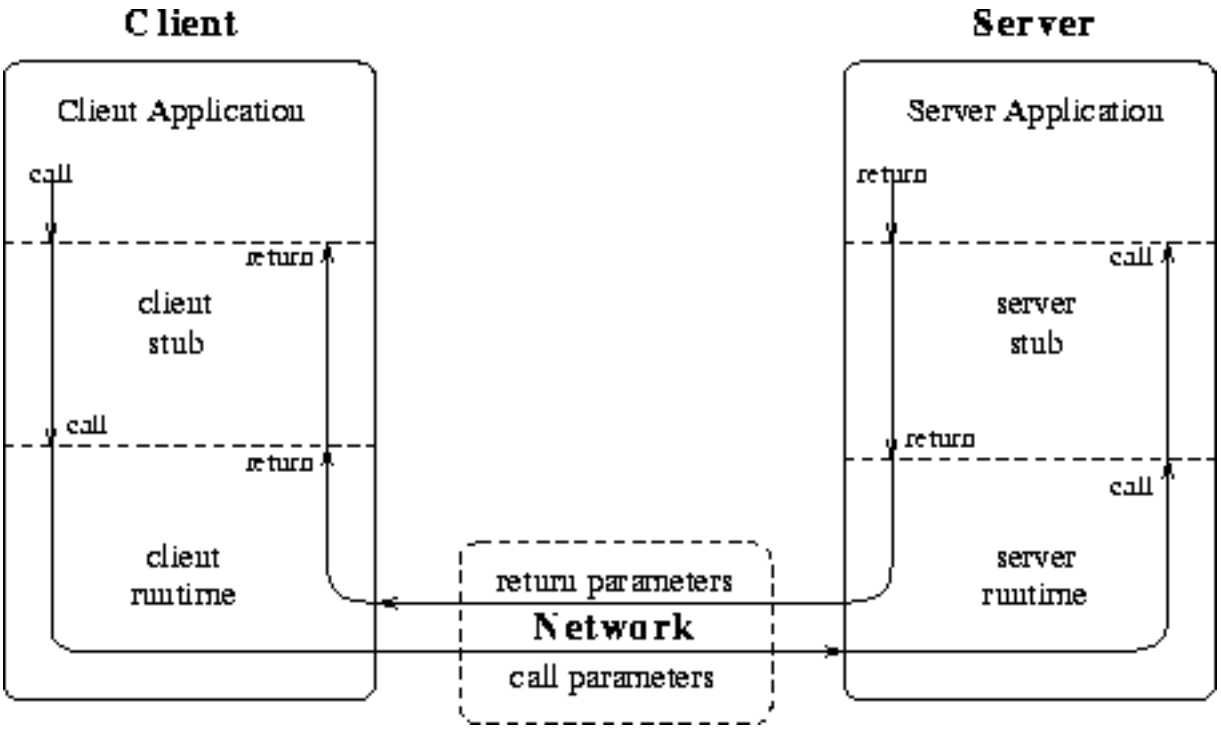
This technique is referred to as *Syscall Proxying* along this document.

A first glimpse into an implementation

When looking at Figure 3 three letters instantly come to my mind: RPC!

Syscall Proxying can be trivially implemented using an RPC model. Each call to an operating system *syscall* is replaced by the corresponding *client stub*. Accordingly, a server that "exports" through the RPC mechanism all these *syscalls* in the remote system is needed (see Figure 4).

Figure 4. The RPC model



There are several RPC implementations that can be used, among these are:

- Open Network Computing (ONC) RPC (also known as SunRPC)
- Distributed Computing Environment (DCE) RPC
- RPC specification from the International Organization for Standardization (ISO)
- Custom implementation

Check Comparing Remote Procedure Calls [http://hissa.nist.gov/rbac/5277/titlerpc.html] for a comparison of the first three implementations.

In the RPC model explained above a lot of effort, symmetrically duplicated between the client and the server, is devoted both to converting back and forth from a common data representation format and to communicating through different calling conventions. These conversions made communication possible between a client and a server implemented in different platforms. Also, the RPC model attempts to attain generality, in making it possible to perform ANY procedure call across a network. If we can move all this logic to the client layer (making the client tightly coupled with the specific server's platform) we can drastically reduce the server's size, taking advantage of a generic mechanism for calling any *syscall* (this mechanism is "generic" when talking about a single platform).

Optimizing for size

Let's say that we have a hunch that there should be some added value in optimizing the *syscall server* size in memory. Also, for simplicity's sake, we'll move into the nice and warm UNIX / i386 world for a couple of lines.

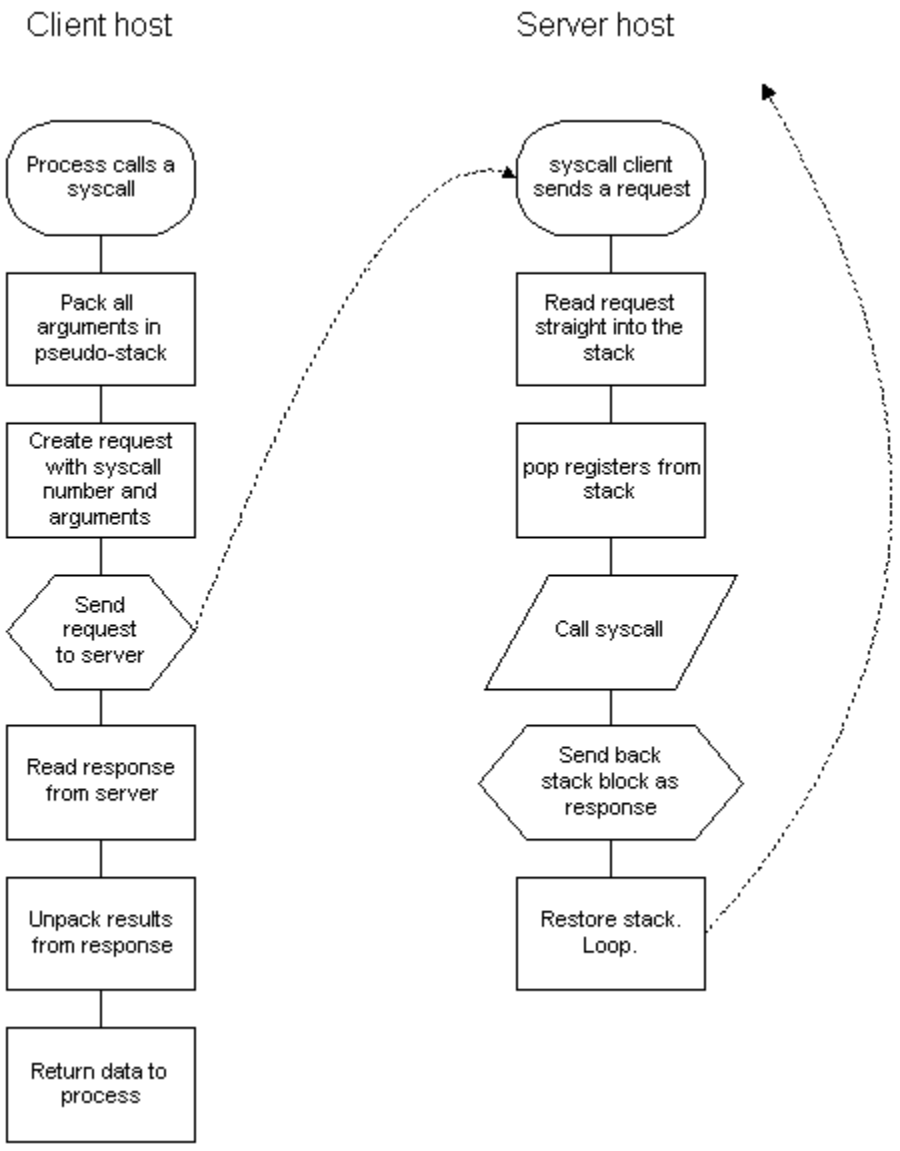
If we look carefully into the *syscalls* we are "proxying" around, they all have a lot in common:

- **Homogeneous way of passing arguments.** *syscall* arguments are either passed through the stack, the registers or a mix of both. Also, these arguments fall into one of these categories:
    - integers
    - pointers to integers
    - pointers to buffers
    - pointers to structs
- **Simple calling mechanism.** A register is set with the number of the *syscall* we'd like to invoke and a generic trap / software interrupt / far call is executed.

Fat client, thin server

In order to accomplish the goal of reducing the *syscall* server's size, we'll make client code do all the work of marshaling each call's arguments directly into the format needed by the server's platform. In fact, the client request will be an image of the server's stack, just before handling the request. In this way, clients will be completely dependent on the server's platform but the *syscall* server will just implement communication, and a generic way of passing the already prepared arguments to the specific *syscall* (see Figure 5).

Figure 5. Fat client, thin server

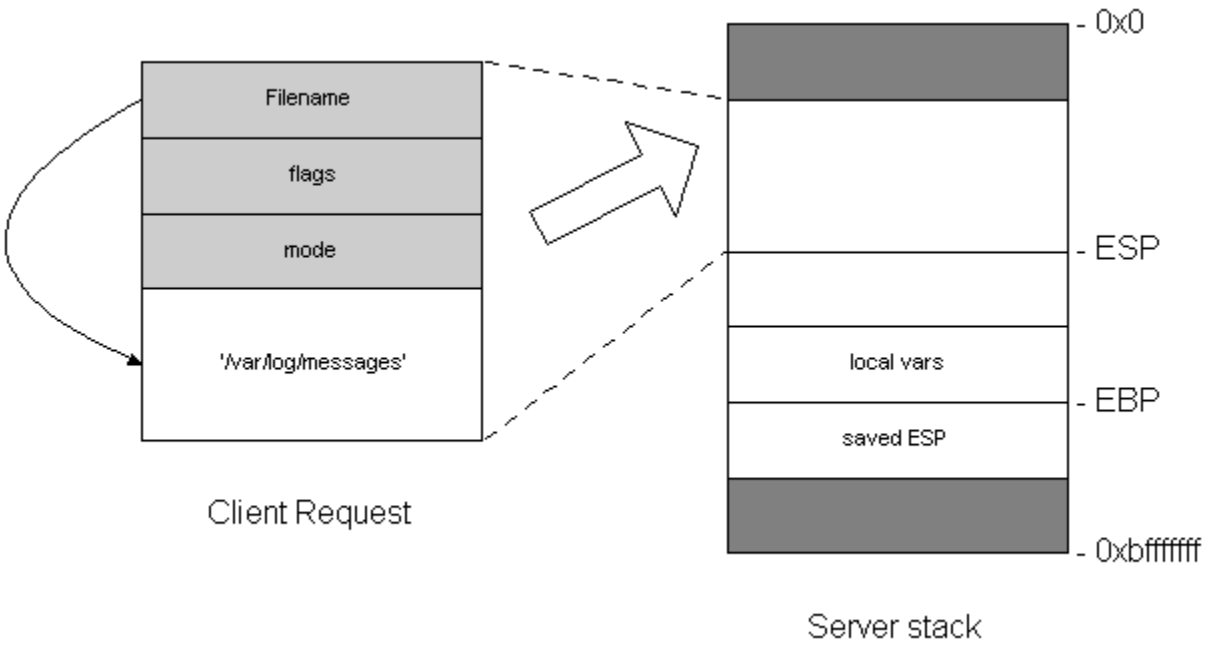


Packing arguments in the client is trivial for integer parameters, but not so for pointers. There's no relationship at all between the client and the server processes memory space. Addresses from the client probably don't make sense in the server and vice versa.

To accommodate for this issue the server, on accepting a new connection from the client, sends back his ESP (the *stack pointer*). The client then creates extra buffer space in the request, and "relocates" each pointer argument to point to somewhere inside the request. Knowing that the server reads requests straight into its stack, the client calculates the correct value for each pointer, as it will be in the server process' stack (this is why we got the server's ESP on the first place).

See Figure 6 for a sample of how `open()` arguments are marshaled.

Figure 6. Marshaling arguments for the open() syscall



A sample implementation for Linux

To invoke a given *syscall* in a i386 Linux system a program has to:

1. Load the EAX register with the desired *syscall* number.

2. Load the *syscall*'s arguments in the `EBX, ECX, EDX, ESI, EDI` registers [2] .

3. Call software interrupt 0x80 (`int $0x80`)

4. Check the returned value in the `EAX` register.

Take a look at Example 1 to see how does the `open()` *syscall* in a Linux system looks when examined through `gdb`.

Example 1. Debugging open() in i386 Linux

```
[max@m21 max]$ gdb ex
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) break open
Breakpoint 1 at 0x8050f60
(gdb) r
Starting program: /usr/home/max/ex

Breakpoint 1, 0x08050f60 in __libc_open ()  ❶
(gdb) l
1       #include <sys/types.h>
2       #include <fcntl.h>
3
4       int main()
5       {
6         int fd;
7
8         fd = open("coconuts", O_RDONLY);
9
10        if(fd<0)
(gdb) x/20i $eip
0x8050f60 <__libc_open>:        push   %ebx
0x8050f61 <__libc_open+1>:      mov    0x10(%esp,1),%edx  ❷
0x8050f65 <__libc_open+5>:      mov    0xc(%esp,1),%ecx  ❸
0x8050f69 <__libc_open+9>:      mov    0x8(%esp,1),%ebx  ❹
0x8050f6d <__libc_open+13>:     mov    $0x5,%eax  ❺
0x8050f72 <__libc_open+18>:     int    $0x80  ❻
```

[2] With some exceptions like the socket functions, where some arguments are passed through the stack.

```
0x8050f74 <__libc_open+20>:     pop    %ebx
0x8050f75 <__libc_open+21>:     cmp    $0xfffff001,%eax
0x8050f7a <__libc_open+26>:     jae    0x8056f50 <__syscall_error>
0x8050f80 <__libc_open+32>:     ret
```

❶ The breakpoint stops execution inside `libc`'s wrapper for the `open()` *syscall*.

❷❸❹ Arguments passed through the stack to `libc_open` are copied into the respective registers.

❺ `EAX` is loaded with 5, the *syscall* number for `open()`.

❻ The software interrupt for system services is triggered.

So, given that a generic and simple mechanism for calling any *syscall* is in place, and using the architecture and argument marshaling techniques explained in the section called "Fat client, thin server", we can code a simple server that looks like Example 2.

Example 2. Pseudocode for a simple Linux server

```
channel = set_up_communication()  ❶
channel.send(ESP)
while channel.has_data() do
   request = channel.read()
   copy request in stack  ❷
   pop registers
   int 0x80
   push eax
   channel.send(stack)  ❸
```

❶ Sets up the communication channel between the client and the server. To keep things simple, we'll assume this is a socket.

❷ Copies the complete request to the server's stack. Remember how does the request look like (see Figure 6).

❸ The request block is sent back to the client. This is done to return any OUT [3] pointer arguments back to the client application. This might be redundant in some cases but since our intention is to keep the server simple, it won't handle these cases differently.

An excerpt from a simple implementation of a *syscall* server implementing this behavior can be seen at Example 3. This excerpt refers only to the main `read request / process / write response` loop.

[3] Arguments used to return information from the *syscall*, as in `read()`.

Example 3. A simple syscall server snippet for i386 Linux

```
                push    %ebx            # fd
read_request:
                mov     %ebp,%esp
                push    %esp
                xor     %eax,%eax
                movl    $4,%edx         # count
read_request2:
                mov     $3,%al          # __NR_read
                mov     %esp,%ecx       # buff
                int     $0x80           # no error checking!
                add     %eax,%ecx       # new buff
                sub     %eax,%edx       # byte to read
                jnz     read_request2   # do while not zero

                pop     %edx            # get number of bytes in packet
                sub     %edx,%esp       # save space for incoming data
read_request3:
                movl    $3,%eax         # __NR_read
                mov     %esp,%ecx       # buff
                int     $0x80           # no error checking  ❶
                add     %eax,%ecx       # new buff
                sub     %eax,%edx       # byte to read
                jnz     read_request3   # do while not zero

do_request:
                pop     %eax  ❷
                pop     %ebx
                pop     %ecx
                pop     %edx
                pop     %esi
                pop     %edi
                int     $0x80  ❸
                push    %edi
                push    %esi
                push    %edx
                push    %ecx
                push    %ebx
                push    %eax

do_send_answer:
                mov     $4,%eax         # __NR_write
                mov     (%ebp),%ebx     # fd
                mov     %esp,%ecx       # buff
```

```
mov     %ebp,%edx
sub     %esp,%edx       # count
int     $0x80           # no error checking!  ❹

jmp     read_request

pop     %eax
pop     %ebp
```

❶ Read request straight into ESP.
❷ Pop registers from the top of the request (the request is in the stack).
❸ Invoke the *syscall*. EAX already holds the *syscall* number (it was part of the request).
❹ Send back the request buffer as response, since buffers might contain data (for example if calling `read()`).

In this way, we are able to code a simple but yet very powerful *syscall* server in a few bytes. The sample complete implementation of the server described above, with absolutely no effort made on optimizing the code, is about a hundred bytes long.

What about Windows?

Yeah! What about it? Ah, yes! Windows is a whole different thing. As explained in the section called “The Windows way”:

> ...we’ll refer to "Windows syscalls" to any function in any dynamic library available to a user mode process.

Even though the boundaries for *syscalls* or the *native API* are not clearly defined, there is a simple and common mechanism for loading any DLL and calling any function in it.

In the same spirit as the Linux server, the windows one will also be based on the techniques explained in the section called “Fat client, thin server”. There’s one subtle difference: in the UNIX world we used an integer to identify which *syscall* to invoke, in the Windows world there’s no such thing (at least not with our definition of windows *syscall*). So, our server will provide functionality for calling any function in its process address space. What does this mean? Initially, that the server can only call functions that are part of whatever DLLs are already loaded by the process. But wait, two special functions are already part of the server’s address space: `LoadLibrary` and `GetProcAddress`.

> The `LoadLibrary` function maps the specified executable module into the address space of the calling process.
>
> (...)

> The `GetProcAddress` function retrieves the address of an exported function or variable from the specified dynamic-link library (DLL).
>
> —*Windows Platform SDK: DLLs, Processes, and Threads*

Using these two functions along with the capability of calling any function in the server's address space, we can fulfill the initial goal of "calling any function on any DLL" (see Example 4).

Example 4. Pseudocode for a simple Windows server

```
channel = set_up_communication()
channel.send(ESP)
channel.send(address of LoadLibrary)      ❶
channel.send(address of GetProcAddress)   ❷
while channel.has_data() do
   request = channel.read()
   copy request in stack
   pop ebx
   call [ebx]                             ❸
   push eax
   channel.send(stack)
```

❶❷ The addresses of `LoadLibrary` and `GetProcAddress` in the server's address space are passed back to the *syscall* client upon initialization. The client can then use the generic "call any address" mechanism implemented by the server's main loop to call these functions and load new DLLs in memory.

❸ The client passes the exact address of the function it wants to call inside the request. The stack is already prepared for the call (as in Figure 6).

In this way, an equivalent server can be coded in the Windows platform. Argument marshaling on the client side is not so trivial, since we are providing a method for calling ANY function. A generic method for "packing" any kind of arguments (integers, pointers, structures, pointers to pointer to structures with pointers, etc) is necessary.

The real world: applications

Syscall Proxying could be used for any task related with remote execution, or with the distribution of tasks among different systems (all goals in common with RPC). The fact is that, for generic remote execution uses, Syscall Proxying (implemented as a non RPC server) lags behind RPC's efficiency and interoperability. But, there's one use in which Syscall Proxying "Fat client, thin server" architecture wins: exploiting code injection vulnerabilities.

Code execution vulnerabilities are those that allow an attacker to execute arbitrary code in the target system. Typical incarnations of code injection vulnerabilities are: buffer overflows and user-supplied format strings. Attacks for these vulnerabilities usually come in 2 parts [4] :

- **Injection Vector (deployment).** The portion of the attack directed at exploiting the specific vulnerability and obtaining control of the instruction pointer (`EIP / PC registers`)..
- **Payload (deployed).** What to execute once we are in control. Not related to the bug at all.

A common piece of code used as attack payload is the "shellcode":

> Shell Code:
>
> So now that we know that we can modify the return address and the flow of execution, what program do we want to execute? In most cases we'll simply want the program to spawn a shell. From the shell we can then issue other commands as we wish. But what if there is no such code in the program we are trying to exploit? How can we place arbitrary instruction into its address space? The answer is to place the code with are trying to execute in the buffer we are overflowing, and overwrite the return address so it points back into the buffer.
>
> —Aleph One, *Smashing The Stack For Fun And Profit*

Shell code allows the attacker to have interactive control of the target system after a successful attack.

The Privilege Escalation phase

The "Privilege Escalation" phase is a critical stage in a typical penetration test. An auditor faces this stage when access to a target host has been obtained, usually by succesfully executing an attack. Once a single host's security has been compromised, either the auditor's goals have been accomplished or they have not.

Access to this intermediate target allows for staging more effective attacks against the system by taking advantage of existing webs of trust between hosts and a more privileged position in the target system's network. When the auditor uses this "vantage point" inside the target's network, he has effectively changed his "attacker profile" from an external attacker to an attacker with access to an internal system.

To successfully "pivot", the auditor needs to be able to use his tools (for info gathering, attack, etc) at the compromised host. This usually involves porting the tools or exploits to a different platform and deploying them on the host, sometimes including installing required libraries and packages and even a C compiler.

These tasks can be accomplished using a remote shell on the compromised system, although they are far from following a formal process, and some times require a lot of effort on the auditor's side.

[4] Taken from Greg Hoglund's, *Advanced Buffer Overflow Techniques*, Black Hat Briefings USA 2000.

Redefining the word "shellcode"

Let's suppose that all our penetration testing tools are developed with Syscall Proxying in mind. They all use a *syscall stub* for communicating with the operating system. Now, if instead of supplying code that spawns a shell as the attack payload, we use a "thin *syscall* server" we can see the following advantages:

- **Transparent pivoting.** Whenever an attack is executed succesfully, a *syscall* server is deployed in the remote host. All of the available tools can now be used in the auditor's host but "proxying" *syscalls* on the newly compromised one. These tools seem to be executing on the remote system.
- **"Local" privilege escalation.** Attacks that obtain remote access with constraints, such as vulnerabilities in services inside *chroot* jails or in services that lower the user privileges can now be further explored. Once a *syscall* server is in place, local exploits / tools can be used to leverage access to higher privileges in the host.
- **No shell? Who cares!** In certain scenarios even though a certain system is vulnerable, it is not possible to execute a shell (picture a *chrooted* service with no root privileges and no local vulnerabilities). In this situation, deploying a *syscall* server still allows the auditor to "proxy" attacks against other systems accessible from the originally compromised system.

Conclusions

Syscall Proxying is a powerful technique when staging attacks against code injection vulnerabilities (buffer overflows, user supplied format strings, etc) to successfully turn the compromised host into a new attack vantage point. It can also come handy when "shellcode" customization is needed for a certain attack (calling `setuid(0)`, deactivating signals, etc).

Syscall Proxying can be viewed as part of a framework for developing new penetration testing tools. Developing attacks that actively use the Syscall Proxying mechanism effectively raises their value.

Acknowledgements

The term "Syscall Proxying" along with a first implementation (a RPC client-server model) was originally brought up by Oliver Friedrichs and Tim Newsham. Later on, Gerardo Richarte and Luciano Notarfrancesco from CORE ST refined the concept and created the first shellcode implementation for Linux.

The CORE IMPACT team worked on a generic syscall abstraction creating the ProxyCall client interface, along with several different server implementations as shellcode for Windows, intel *BSD and SPARC Solaris. The ProxyCall interface constitutes the basis for IMPACT's multi-platform module framework and they are the basic building block for IMPACT's agents.

"Pivoting" was coined by Ivan Arce in mid 2001.