Syscall Proxying | Simulating Remote Execution

Maximiliano Cáceres | maximiliano.caceres@corest.com

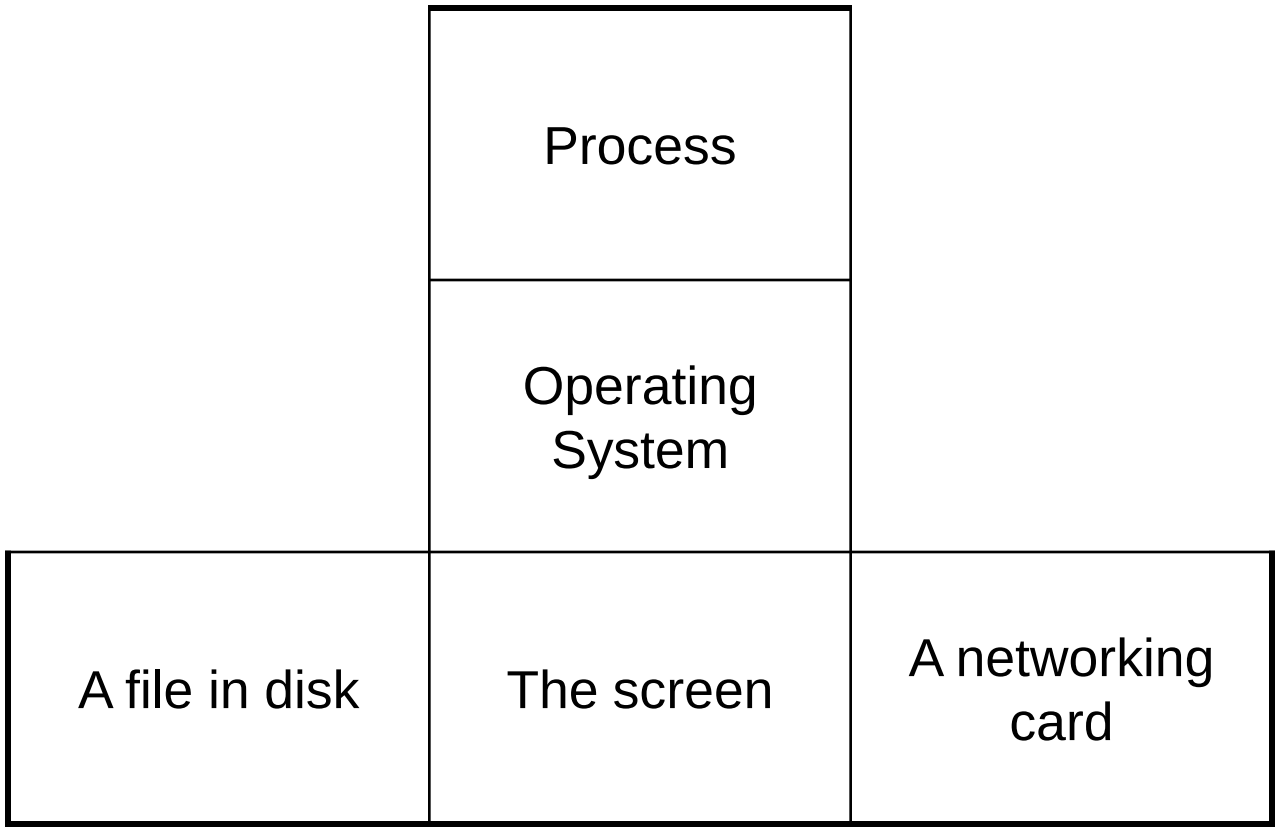
Agenda

- General Concepts
- Syscall Proxying
- A first implementation
- Optimizing for size
- The real world: applications

Agenda

General Concepts

A Process Interacts with Resources



Syscalls

- ***System calls (aka "syscalls")***
  - Operating system services
  - Lowest layer of communication between a user mode process and the kernel

The System Services Layer

The UNIX Way

- Homogeneous mechanism for calling any syscall by number
- Arguments passed through the stack or registers
- Minimum number of system services
- Direct mapping between syscall and libc wrapper

The Windows Way

- *Native API* undocumented and unsupported
- High number of system level services (about 1000)
- Win32 API calls implement a lot of functionality around these services

Our Windows "Syscalls"

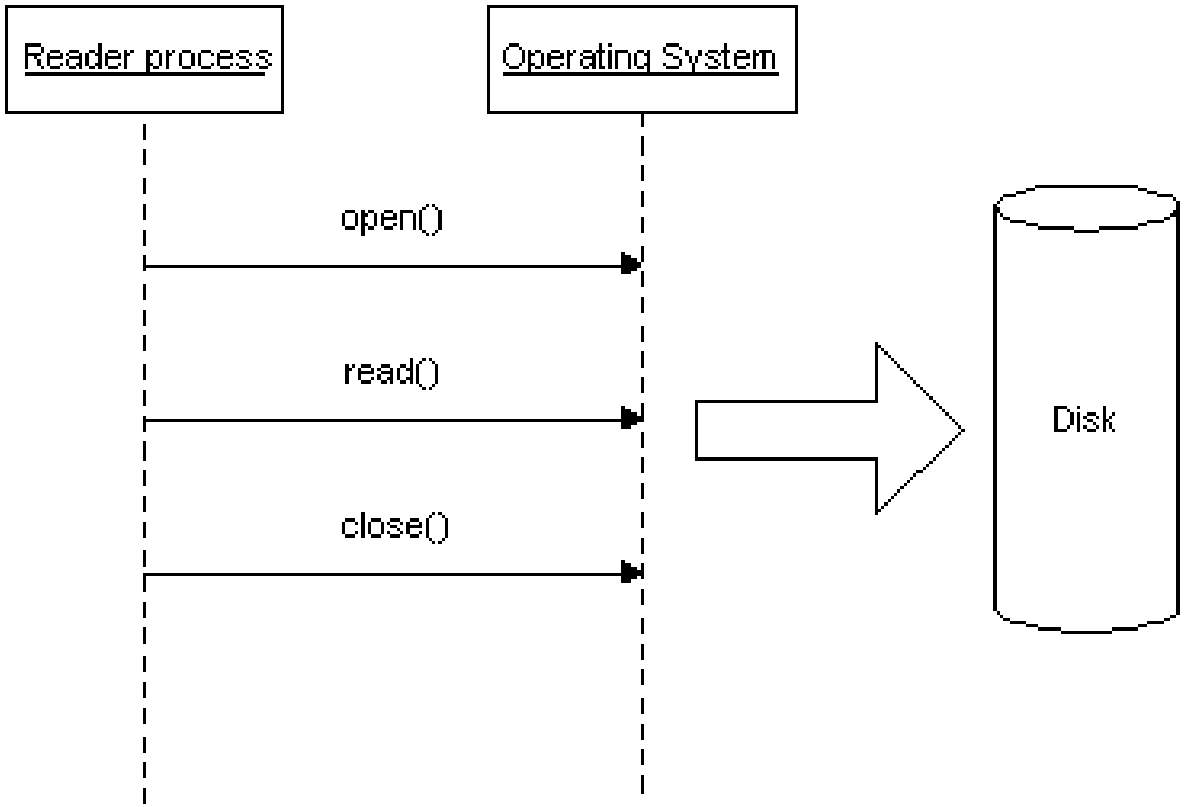
- ***Keep things simple***
  - ANY function in ANY dynamic library available to a user mode process

Syscall Proxying

The Process "Context"

- ***A process uses resources to accomplish a goal***
- ***These resources define a "context" on which the process runs***
  - The specific resource instances
  - The kind of access to these resources

A process reading data from a file



Two Layers

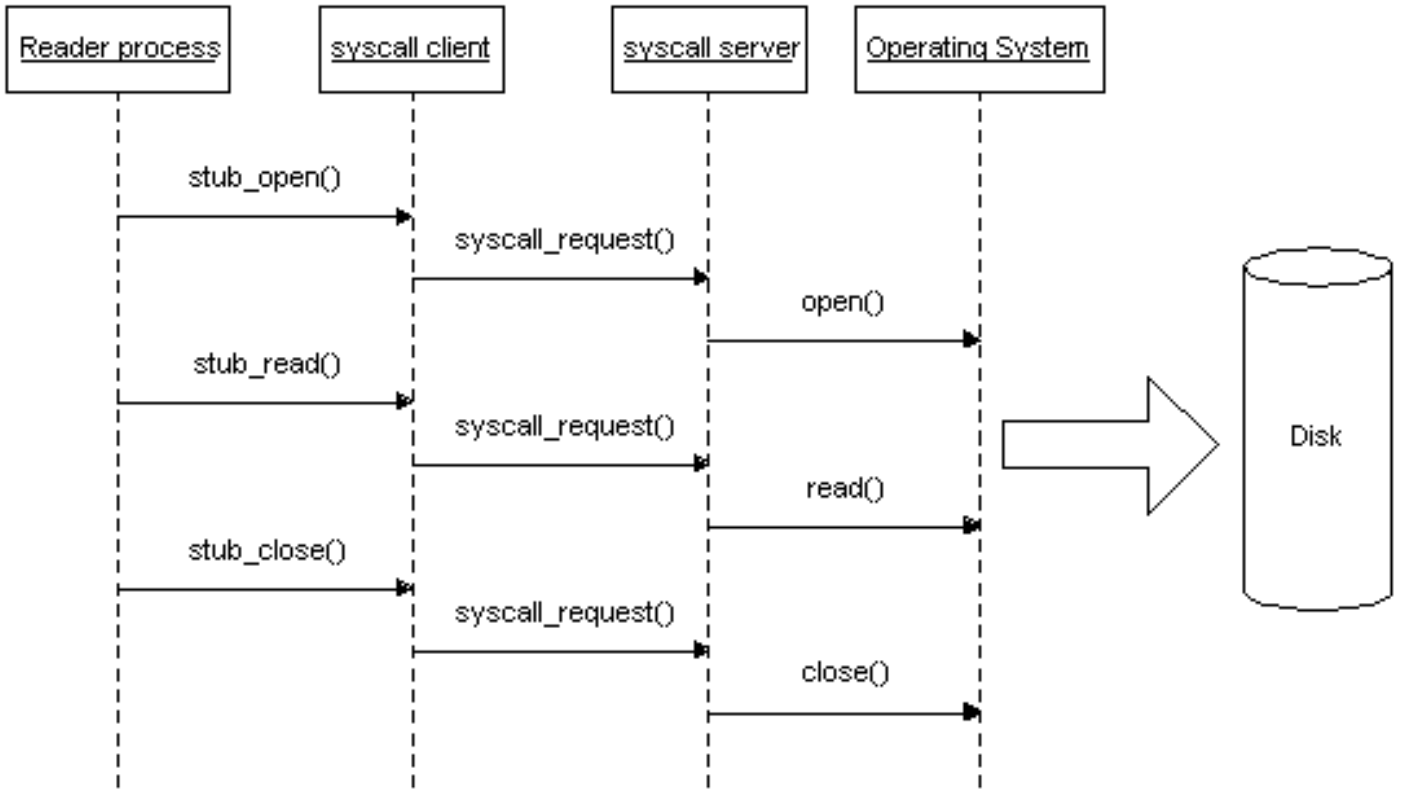
Syscall stub / client

- Nexus between process and system services
- Converts syscall argument to a common format (marshaling)
- Sends requests to the syscall server
- Marshals return values

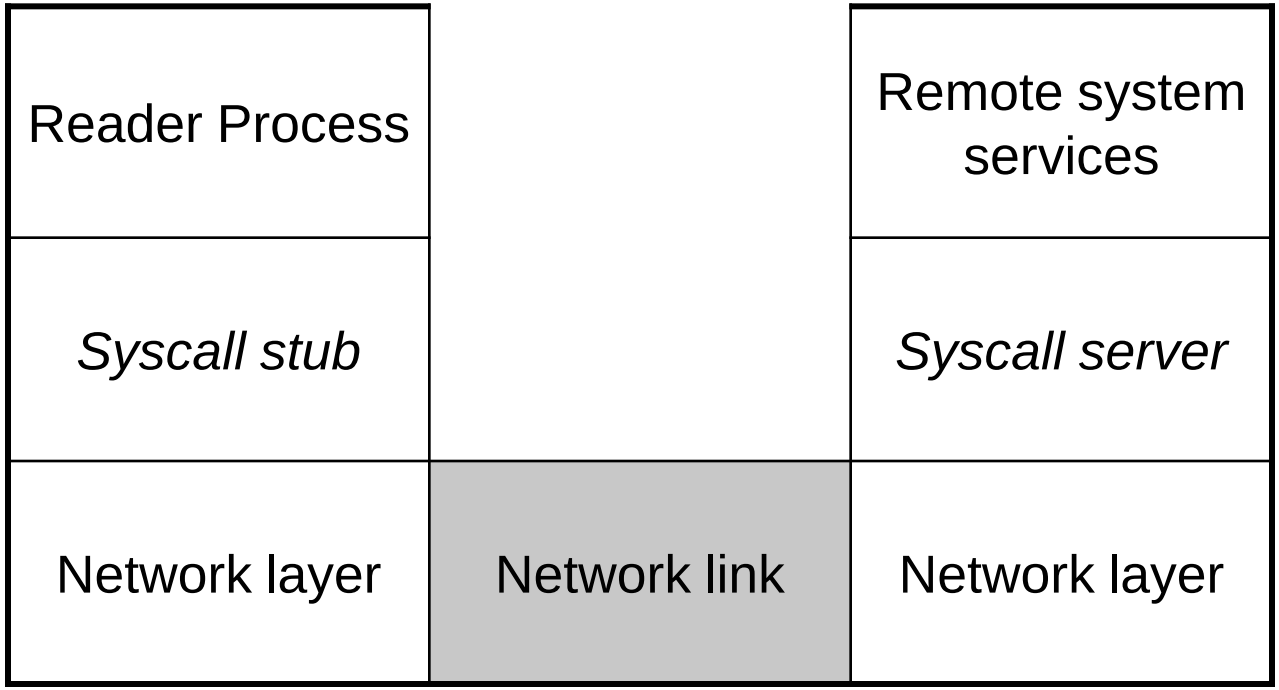
Syscall server

- Handles requests from the syscall client
- Converts arguments in request to native convention
- Calls the specified syscall
- Sends back a response to the client

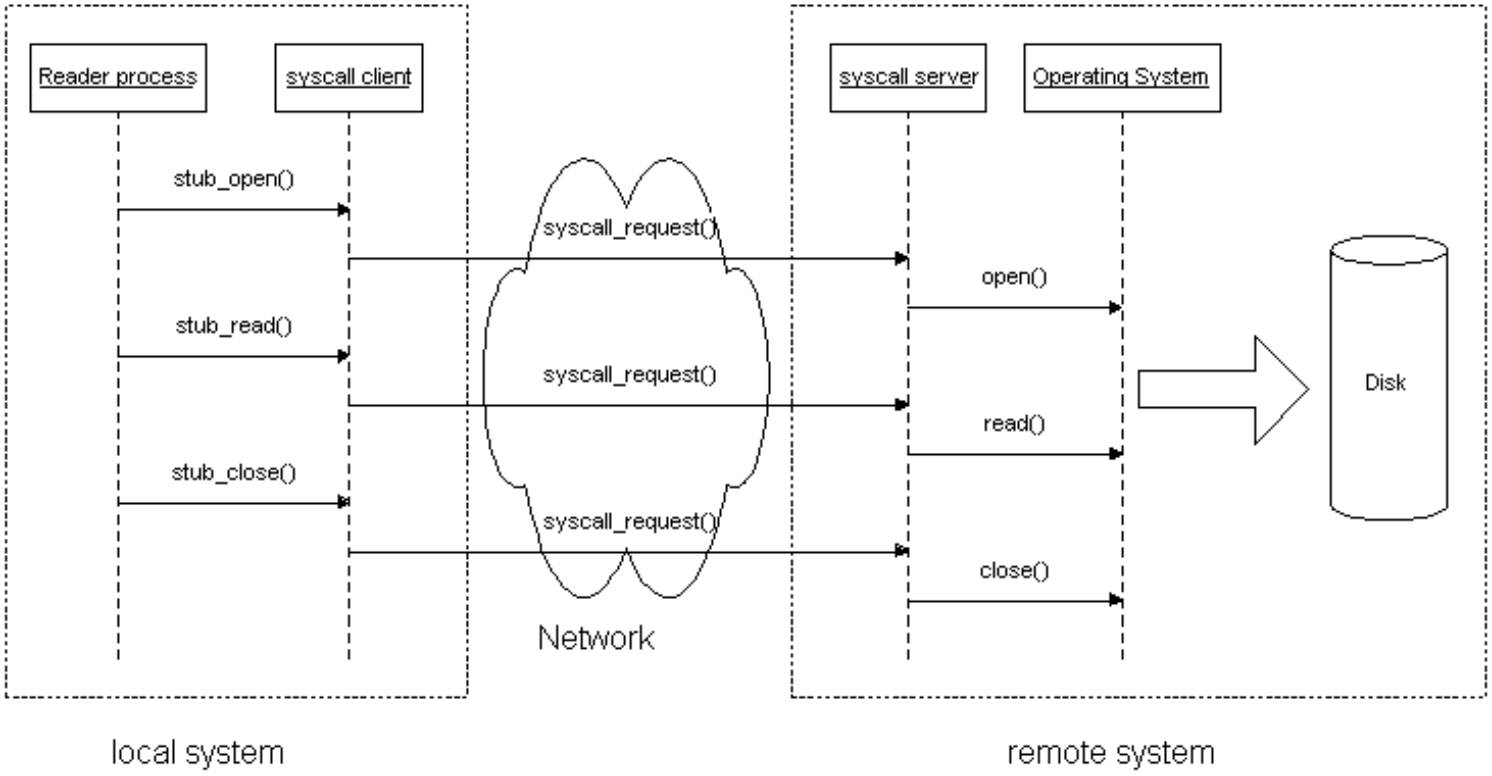
A process reading data from a file, using these two layers



Separating Client from Server



Syscall Proxying in Action



Changing Context

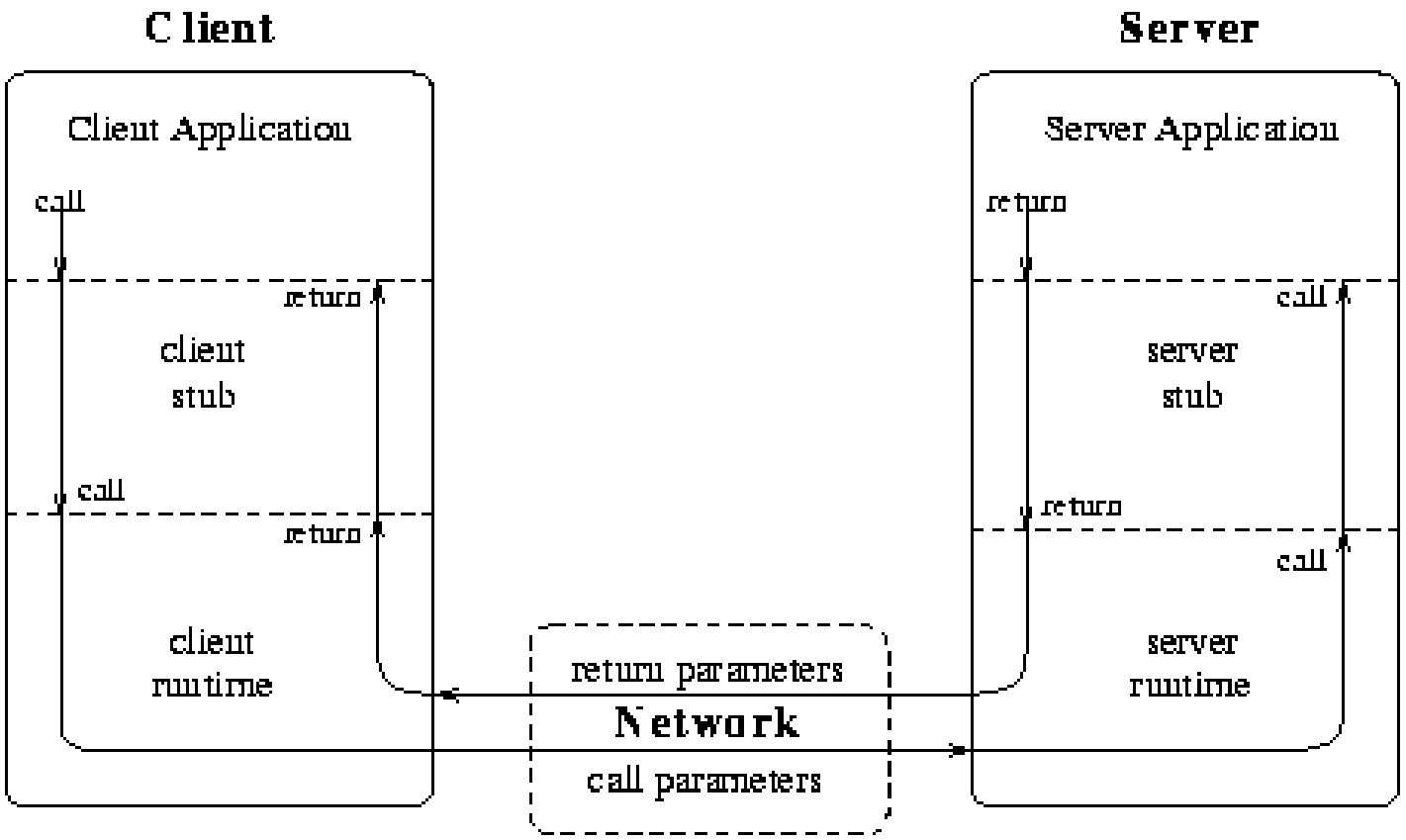
- ***Separating client from server***
  - The process accesses remote resources (a file)
  - The process uses the privileges of the remote server
  - The process doesn't now anything about remote execution
- ***No modifications on the original program***
  - Same inner logic

A first implementation

Implementing Syscall Proxying

- ***The RPC Model***
  - Client / server
  - Remote calls are handled by both a client stub and a server stub
- ***Perfect match!***

The RPC Model



Benefits and Shortcomings of the RPC Model

- ***Benefits***
  - Interoperability between different platforms
  - Almost any procedure call can be converted to RPC
- ***Shortcomings***
  - Both client and server symmetrically duplicate data conversion to a common data interchange format

Optimizing for size

The UNIX *Syscall* Mechanism

- ***Homogeneous way of passing arguments***
  - Integers
  - Pointers to integers
  - Pointers to buffers
  - Pointers to structs
- ***Simple calling mechanism***
  - Software interrupt
  - Trap
  - Far call

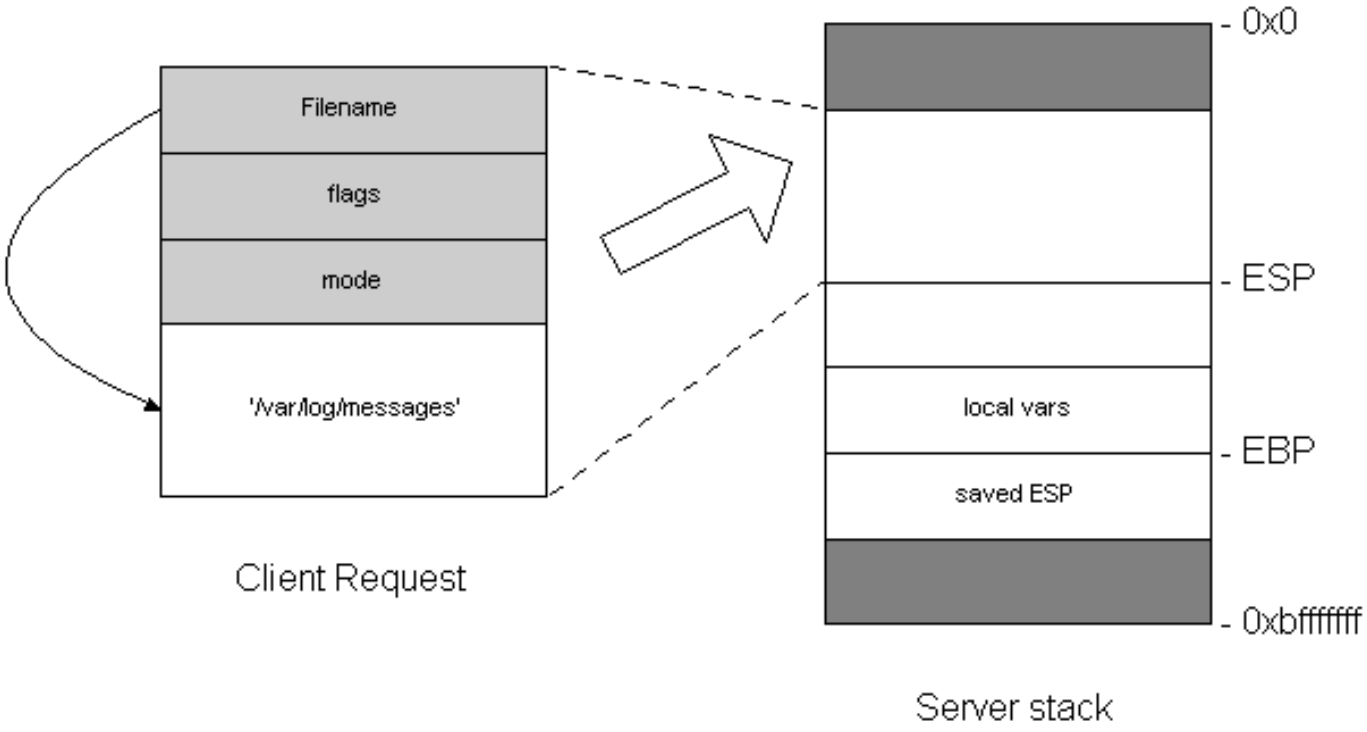
Fat Client, Thin Server

- *Client code directly converts from the client system's calling convention to the server's (no intermediate common format)*
- *The server takes advantage of the generic mechanism for calling syscalls*
- *The client is completely dependent on the server's platform*

Marshaling Arguments

- ***Client code creates a request representing the stack state in the server just before invoking the syscall***
  - Integers are trivially packed
  - Pointers to buffers or structures are relocated inside the same request buffer using the server's stack pointer

Marshaling arguments for open()



Linux syscalls

- ***Invoking a syscall in Linux***
  - Load EAX with syscall number
  - Load arguments in EBX, ECX, EDX, ESI and EDI (syscalls with more than 5 arguments push the rest on the stack)
  - Call software interrupt 0x80 (int $0x80)
  - Return value in EAX

Debugging open()

```
Breakpoint 1, 0x08050f60 in __libc_open ()
(gdb) x/20i $eip
<__libc_open>: push %ebx
0x8050f61 <__libc_open+1>: mov 0x10(%esp,1),%edx
0x8050f65 <__libc_open+5>: mov 0xc(%esp,1),%ecx
0x8050f69 <__libc_open+9>: mov 0x8(%esp,1),%ebx
0x8050f6d <__libc_open+13>: mov $0x5,%eax
0x8050f72 <__libc_open+18>: int $0x80
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
```

A simple Linux server

Pseudocode for a simple linux server

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

A simple syscall server in Linux (1)

- ***Read request straight into the stack***

```
read_request:
                mov     fd, %ebx
                mov     buflen, %edx
                movl    $3,%eax          # __NR_read
                mov     %esp,%ecx       # buff
                int     $0x80
```

A simple syscall server in Linux (2)

- ***Invoke the desired syscall***

```
do_request:
                    pop     %eax
                    pop     %ebx
                    pop     %ecx
                    pop     %edx
                    pop     %esi
                    pop     %edi
                    int     $0x80
```

  - The request previously stored in ESP is the stack needed by the syscall PLUS buffers

A simple syscall server in Linux (3)

- *Coding a simple syscall server for Linux can be done*
- *It takes about a hundred bytes long (without optimizing)*

What about Windows?

- ***Windows "syscalls"***
  - *"… any function in any dynamic library available to a user mode process."*
  - Common mechanism

The Windows Syscall Server (1)

- ***Windows server***
  - Call any function in its process address space (already loaded)
- ***In particular***
  - Call LoadLibrary to load a new DLL
  - Call GetProcAddress to obtain the address of a specific function

The Windows Syscall Server (2)

- ***Pseudocode for a sample Windows server***

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

The Real World: applications

Exploiting Code Injection Vulnerabilities

- ***Allow an attacker to execute arbitrary code in the target system***
  - Buffer overflows
  - User supplied format strings
- ***Attack method***
  - Injection: attack specific
  - Payload: what to execute once control is gained
  - Shell code: code that spawns a shell

The Privilege Escalation Phase

- ***Successful attack against a host.***
- ***Use the compromised host as vantage point ("pivoting")***
  - Attacker profile switch: from external to internal
  - Exploit webs of trust
  - Possibly more privileged position in the target system's network
- ***To be able to "pivot", the auditor needs his tools available at the vantage point***

Redefining the word "shellcode"

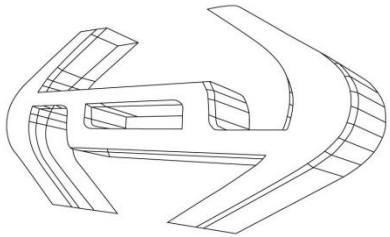
- ***Supply "thin" syscall server as attack payload***
- ***Benefits***
  - Transparent pivoting
  - "Local" privilege escalation
  - No shell? Who cares!

Conclusions

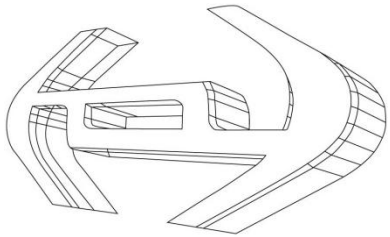
Conclusions

- ***Powerful technique when staging attacks against code injection vulnerabilities***
  - Turns the compromised host into a new attack vantage point
  - Useful when shell code customization is needed
- ***Framework for developing new penetration testing tools***
  - Raises the value of the tools

Questions?



Thank You!



- **Maximiliano Cáceres** | maximiliano.caceres@corest.com

CORE SECURITY TECHNOLOGIES · Offices Worldwide

Headquarters
44 Wall Street | 12th Floor
New York, NY 10005 | USA
Ph: (212) 461-2345
Fax: (212) 461-2346
info.usa@corest.com

Florida 141 | 2º cuerpo | 7º piso
(C1005AAC) Buenos Aires
Tel/Fax: (54 11) 4878-CORE (2673)
info.argentina@corest.com

Rua do Rócio 288 | 7º andar
Vila Olímpia | São Paulo | SP
CEP 04552-000 | Brazil
Tel: (55 11) 3054-2535
Fax: (55 11) 3054-2534
info.brazil@corest.com

www.corest.com