CVE-2018-6789

Mod233

July 20, 2018

Abstract

这是我调试的第一个 CVE，总的来说漏洞难度适中，非常适合入门的我。

CVE-2018-6789 是一个针对 Linux 下邮件服务软件 exim 的 RCE 漏洞，溢出点在于没有对 Base64 解密的输入进行检查，造成的 off-by-one 漏洞。

Contents

1 漏洞分析

这个章节主要讲下 GDB 插件的配置，及 GDB 中，设置断点、堆栈查看、代码查看等命令的使用。

2 调试笔记

为了方便，选用别人给出的 docker 虚拟机进行调试：

```
 ~ docker run -it --name exim -p 25:25 skysider/vulndocker:cve-2018-6789
# other terminal windows run:
 ~ docker exec -it 403284d44dd2 /bin/bash
root@403284d44dd2:/work#
# check the pid:
 ~ docker exec -it 403284d44dd2 /bin/bash
root@403284d44dd2:/work# ps -aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
exim          1  0.6  0.0  29948  3552 pts/0    Ss+  10:49   0:00 /work/exim-4.89
root          7  0.0  0.0  21480  3560 pts/1    Ss   10:49   0:00 /bin/bash
root         22  0.0  0.0  37656  3268 pts/1    R+   10:49   0:00 ps -aux

# use gdb to attach the process
```

说真的，别人的流程都多多少少都遗漏了一些关键步骤，这里我决定把这次这个漏洞尽可能钻研的深一些，不仅是漏洞钻研的深一些，整个一套体系，比如 *docker*，*git checkout*，*gdb* 这些，都尽可能深入一些。

既然别人用了 Docker，那么我就详细看看别人的 dockerfile：

```
FROM phusion/baseimage:latest
LABEL maintainer=skysider@163.com

RUN apt-get -y update && \
    DEBIAN_FRONTEND=noninteractive apt-get install -y \
    wget \
```

```
    xz-utils \
    make \
    gcc \
    libpcre++-dev \
    libdb-dev \
    libxt-dev \
    libxaw7-dev \
    tzdata \
    telnet && \
    rm -rf /var/lib/apt/lists/*

WORKDIR /work

COPY conf.conf /work/

RUN wget https://github.com/Exim/exim/releases/download/exim-4_89/exim-4.89.tar.xz && \
    tar xf exim-4.89.tar.xz && cd exim-4.89 && \
    cp src/EDITME Local/Makefile && cp exim_monitor/EDITME Local/eximon.conf && \
    sed -i 's/# AUTH_CRAM_MD5=yes/AUTH_CRAM_MD5=yes/' Local/Makefile && \
    sed -i 's/^EXIM_USER=/EXIM_USER=exim/' Local/Makefile && \
    useradd exim && make && mkdir -p /var/spool/exim/log && \
    cd /var/spool/exim/log && touch mainlog paniclog rejectlog && \
    chown exim mainlog paniclog rejectlog && \
    echo "Asia/Shanghai" > /etc/timezone && \
    cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

CMD ["/work/exim-4.89/build-Linux-x86_64/exim", "-bd", "-d-receive", "-C", "conf.conf"]
```

显而易见的几点：

1. FROM：代表镜像的源头。
2. RUN：代表要执行的命令，跟 shell 脚本类似。
3. WORDIR：用于切换工作路径。
4. CMD：这个以‘,’分隔开，每个双引号中代表一个命令参数。

因为涉及到调试，那么一定要去看看 Makefile，我现在还很纠结的，如何自己构建 Makefile，并且将自己的调试信息放在里面。

```
root@403284d44dd2:/work/exim-4.89# cat Makefile |grep -v "#"

SHELL=/bin/sh
RM_COMMAND=/bin/rm

buildname=$${build:-`$(SHELL) scripts/os-type`-`$(SHELL) scripts/arch-type`}$${EXIM_BUILD_SUFFIX:+.$$EXIM_BUILD_SUFFIX}

all: Local/Makefile configure
	@cd build-$(buildname); $(MAKE) SHELL=$(SHELL) $(MFLAGS)
exim: Local/Makefile configure
	@cd build-$(buildname); $(MAKE) SHELL=$(SHELL) $(MFLAGS) exim
utils: Local/Makefile configure
	@cd build-$(buildname); $(MAKE) SHELL=$(SHELL) $(MFLAGS) utils

Local/Makefile:
	@echo ""
	@echo "*** Please create Local/Makefile by copying src/EDITME and making"
	@echo "*** appropriate changes for your site."
	@echo ""
	@test ! -d Local && mkdir Local
	@false

build-directory:
	@builddir=build-$(buildname); \
	case "$$builddir" in *UnKnown*) exit 1;; esac; \
	$(SHELL) -c "test -d $$builddir -a -r $$builddir/version.c || \
	(mkdir $$builddir; cd $$builddir; $(SHELL) ../scripts/MakeLinks)";
checks:
```

```
29      $(SHELL) scripts/source_checks
30
31  configure: checks build-directory
32      @cd build-$(buildname); \
33        build=$(build) $(SHELL) ../scripts/Configure-
              Makefile
34
35  makefile: build-directory
36      @cd build-$(buildname); $(RM_COMMAND) -f Makefile; \
37        build=$(build) $(SHELL) ../scripts/Configure-
              Makefile
38
39  install:        all
40          @cd build-$(buildname); \
41          build=$(build) $(SHELL) ../scripts/exim_install $(
              INSTALL_ARG)
42
43  clean:; @echo ""; echo '*** "make␣clean" just removes all
       .o and .a files'
44      @echo '*** Use "make␣makefile" to force a rebuild of
           the makefile'
45      @echo ""
46      cd build-$(buildname); \
47      $(RM_COMMAND) -f *.o lookups/*.o lookups/*.a auths/*.o
           auths/*.a \
48      routers/*.o routers/*.a transports/*.o transports/*.a
           \
49      pdkim/*.o pdkim/*.a
50
51  clean_exim:; cd build-$(buildname); \
52       $(RM_COMMAND) -f *.o lookups/*.o lookups/*.a auths/*.
           o auths/*.a \
53      routers/*.o routers/*.a transports/*.o transports/*.a
           lookups/*.so
54
55  distclean:; $(RM_COMMAND) -rf build-* cscope*
56
57  cscope.files: FRC
58      echo "-q" > $@
59      echo "-p3" >> $@
```

```
60      find src Local OS exim_monitor -name "*.[cshyl]" -
            print \
61              -o -name "os.h*" -print \
62              -o -name "*akefile*" -print \
63              -o -name config.h.defaults -print \
64              -o -name EDITME -print >> $@
65      ls OS/* >> $@
66
67  FRC:
```

真是醉了，智障一样看了这么久的 Makefile、文件结构找不到突破点，后来想当然动手一下就成功了... 注意看上面的 Makefile 的中，有一个 *MFLAGS* 的变量，但却找不到定义，我尝试对其进行输出，也没有结果。后来我随心的一次编译失败：

```
1  ...SNIP...
2  SHELL=/bin/sh
3  RM_COMMAND=/bin/rm
4  MFLAGS=-d
5  ..SNIP...
```

或者使用：

```
1  Usage: make [options] [target] ...
2  Options:
3    -b, -m                      Ignored for compatibility.
4    -B, --always-make           Unconditionally make all
       targets.
5    -C DIRECTORY, --directory=DIRECTORY
6                                Change to DIRECTORY before
                                   doing anything.
7    -d                          Print lots of debugging
       information.
8    --debug[=FLAGS]             Print various types of
       debugging information.
9    -e, --environment-overrides
10
11 make -d
```

这样命令行中的参数就会传递给 *MFLAGS*。

```
root@403284d44dd2:/work/exim-4.89# make
...SNIP...
Reaping winning child 0x21d4310 PID 8336
Removing child 0x21d4310 PID 8336 from chain.
  Successfully remade target file 'exim'.
 Finished prerequisites of target file 'all'.
Must remake target 'all'.
Successfully remade target file 'all'.
make[1]: Leaving directory '/work/exim-4.89/build-Linux-
   x86_64'
buildname is Linux-x86_64
mflags is -d
shell is /bin/sh
make is make
```

编译了带调试信息后可执行文件后，可以利用 *nm* 命令看下其中的符号标记：

```
  build-Linux-x86_64 nm exim|grep "[tT] "
00000000006b97d8 D accept_8bitmime
                 U accept@@GLIBC_2.2.5
00000000006bdd4c b accept_retry_count
00000000006bdd48 b accept_retry_errno
00000000006bdd44 b accept_retry_select_failed
0000000000471c95 T accept_router_entry
...SNIP...
00000000006b6bb8 d write_pid
00000000006b8780 D write_rejectlog
0000000000438f8f t write_syslog
00000000004394e2 T write_to_fd_buf
000000000046b4ea t write_tree
                 U __xstat64@@GLIBC_2.2.5
00000000006bfbd8 b yield.16248
00000000006bbaf0 d yield_length
```

所有的符号标记，都是可以下断点的，并且：

U：代表未定义符号，通常为外部符号引用
T：代表在文本中定义的符号，通常为函数名称
t：在文本部分定义的局部符号，在 C 函数中通常是某个函数
D：已初始化的数据值

C：未初始化的数据值

默认情况下，Core 配置是关闭的，因为 Core 文件会拖慢程序崩溃后重启的速度，并设计敏感信息。通过 *ulimit c* 命令和*/procsys/kernel/core_pattern* 文件能够得到当前 core 文件的配置情况：

2.1 Core 调试

完成了 core 的配置，并通过 GDB 进入后，下一步就是要提取 core 中的信息来查看程序崩溃时堆栈、函数调用的情况了，下面的例子我选用的都是别人的输出，因为别人选择的 core 报警相对复杂，可以看到更复杂的输出，一般步骤为：

1. 利用 *backtrace* 命令回溯系统栈对函数的调用情况。

```
(gdb) bt
#0  0x00007f0a37aac40d in doupdate () from /lib/x86_64-
    linux-gnu/libncursesw.so.5
#1  0x00007f0a37aa07e6 in wrefresh () from /lib/x86_64-
    linux-gnu/libncursesw.so.5
#2  0x00007f0a37a99616 in ?? () from /lib/x86_64-linux-gnu
    /libncursesw.so.5
#3  0x00007f0a37a9a325 in wgetch () from /lib/x86_64-linux
    -gnu/libncursesw.so.5
#4  0x00007f0a37cc6ec3 in ?? () from /usr/lib/python2.7/
    lib-dynload/_curses.x86_64-linux-gnu.so
#5  0x00000000004c4d5a in PyEval_EvalFrameEx ()
#6  0x00000000004c2e05 in PyEval_EvalCodeEx ()
#7  0x00000000004def08 in ?? ()
#8  0x00000000004b1153 in PyObject_Call ()
#9  0x00000000004c73ec in PyEval_EvalFrameEx ()
#10 0x00000000004c2e05 in PyEval_EvalCodeEx ()
#11 0x00000000004caf42 in PyEval_EvalFrameEx ()
#12 0x00000000004c2e05 in PyEval_EvalCodeEx ()
#13 0x00000000004c2ba9 in PyEval_EvalCode ()
#14 0x00000000004f20ef in ?? ()
#15 0x00000000004eca72 in PyRun_FileExFlags ()
#16 0x00000000004eb1f1 in PyRun_SimpleFileExFlags ()
#17 0x000000000049e18a in Py_Main ()
```

```
#18 0x00007f0a3be10830 in __libc_start_main (main=0x49daf0
    <main>, argc=2, argv=0x7ffd33d94838, init=<optimized
   out>, fini=<optimized out>, rtld_fini=<optimized out>,
    stack_end=0x7ffd33d94828) at ../csu/libc-start.c:291
#19 0x000000000049da19 in _start ()
```

查看的函数栈的时候从下往上的顺序，如果中途出现”??“，一般是”symbol translation failed“。遇到这种情况时，可以找一些 gdb 的插件，或者在 gcc 编译的时候，保留符号信息（*-fno-omit-frame-pointer -g*）来修复这些问题。

具体看下上面的输出，从 frames 5 to 17 都是与 python 相关的调用，尽管不确定具体的 modules 调用情况，但基本的脉络为：*wgetch()->wrefresh()->doupdate()*，接下来就需要对栈中最顶层的 *doupdate()* 进行分析。

2. 利用 *disas func* 命令反汇编函数栈最上层函数。

```
(gdb) disas doupdate
Dump of assembler code for function doupdate:
   0x00007f0a37aac2e0 <+0>:    push   %r15
   0x00007f0a37aac2e2 <+2>:    push   %r14
   0x00007f0a37aac2e4 <+4>:    push   %r13
   0x00007f0a37aac2e6 <+6>:    push   %r12
   0x00007f0a37aac2e8 <+8>:    push   %rbp
   0x00007f0a37aac2e9 <+9>:    push   %rbx
   0x00007f0a37aac2ea <+10>:   sub    $0xc8,%rsp
[...]
---Type <return> to continue, or q <return> to quit---
[...]
   0x00007f0a37aac3f7 <+279>: cmpb   $0x0,0x21(%rcx)
   0x00007f0a37aac3fb <+283>: je     0x7f0a37aacc3b <
      doupdate+2395>
   0x00007f0a37aac401 <+289>: mov    0x20cb68(%rip),%rax
              # 0x7f0a37cb8f70
   0x00007f0a37aac408 <+296>: mov    (%rax),%rsi
   0x00007f0a37aac40b <+299>: xor    %eax,%eax
=> 0x00007f0a37aac40d <+301>: mov    0x10(%rsi),%rdi
   0x00007f0a37aac411 <+305>: cmpb   $0x0,0x1c(%rdi)
   0x00007f0a37aac415 <+309>: jne    0x7f0a37aac6f7 <
      doupdate+1047>
   0x00007f0a37aac41b <+315>: movswl 0x4(%rcx),%ecx
```

```
   0x00007f0a37aac41f <+319>: movswl 0x74(%rdx),%edi
   0x00007f0a37aac423 <+323>: mov    %rax,0x40(%rsp)
[...]
```

只输入 *disas* 命令也会默认的反汇编栈帧中最顶层的函数。标示”=>“代表出错执行的指令。根据这条指令，可以将错误定位到寄存器，下面查看寄存器的值即可。3. 利用 *info registers* 命令查看寄存器的值。

```
(gdb) i r
rax            0x0  0
rbx            0x1993060    26816608
rcx            0x19902a0    26804896
rdx            0x19ce7d0    27060176
rsi            0x0  0
rdi            0x19ce7d0    27060176
rbp            0x7f0a3848eb10   0x7f0a3848eb10 <SP>
rsp            0x7ffd33d93c00   0x7ffd33d93c00
r8             0x7f0a37cb93e0   139681862489056
r9             0x0  0
r10            0x8  8
r11            0x202    514
r12            0x0  0
r13            0x0  0
r14            0x7f0a3848eb10   139681870703376
r15            0x19ce7d0    27060176
rip            0x7f0a37aac40d   0x7f0a37aac40d <doupdate
   +301>
eflags         0x10246  [ PF ZF IF RF ]
cs             0x33 51
ss             0x2b 43
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
```

可以看到，%rsi 的值为 0 ，很明显 0x0 不是一个有效的地址空间，出现了一种常见 segfault: *dereferencing an uninitialized or NULL pointer*.

4. 利用 *i proc m* 命令检查内存分配情况。

```
(gdb) i proc m
Mapped address spaces:

      Start Addr           End Addr       Size     Offset
          objfile
        0x400000           0x6e7000   0x2e7000        0x0
           /usr/bin/python2.7
        0x8e6000           0x8e8000     0x2000   0x2e6000
           /usr/bin/python2.7
        0x8e8000           0x95f000    0x77000   0x2e8000
           /usr/bin/python2.7
  0x7f0a37a8b000     0x7f0a37ab8000    0x2d000        0x0
     /lib/x86_64-linux-gnu/libncursesw.so.5.9
  0x7f0a37ab8000     0x7f0a37cb8000   0x200000    0x2d000
     /lib/x86_64-linux-gnu/libncursesw.so.5.9
  0x7f0a37cb8000     0x7f0a37cb9000     0x1000    0x2d000
     /lib/x86_64-linux-gnu/libncursesw.so.5.9
  0x7f0a37cb9000     0x7f0a37cba000     0x1000    0x2e000
     /lib/x86_64-linux-gnu/libncursesw.so.5.9
  0x7f0a37cba000     0x7f0a37ccd000    0x13000        0x0
     /usr/lib/python2.7/lib-dynload/_curses.x86_64-linux-
     gnu.so
  0x7f0a37ccd000     0x7f0a37ecc000   0x1ff000    0x13000
     /usr/lib/python2.7/lib-dynload/_curses.x86_64-linux-
     gnu.so
  0x7f0a37ecc000     0x7f0a37ecd000     0x1000    0x12000
     /usr/lib/python2.7/lib-dynload/_curses.x86_64-linux-
     gnu.so
  0x7f0a37ecd000     0x7f0a37ecf000     0x2000    0x13000
     /usr/lib/python2.7/lib-dynload/_curses.x86_64-linux-
     gnu.so
  0x7f0a38050000     0x7f0a38066000    0x16000        0x0
     /lib/x86_64-linux-gnu/libgcc_s.so.1
  0x7f0a38066000     0x7f0a38265000   0x1ff000    0x16000
     /lib/x86_64-linux-gnu/libgcc_s.so.1
  0x7f0a38265000     0x7f0a38266000     0x1000    0x15000
     /lib/x86_64-linux-gnu/libgcc_s.so.1
  0x7f0a38266000     0x7f0a3828b000    0x25000        0x0
     /lib/x86_64-linux-gnu/libtinfo.so.5.9
  0x7f0a3828b000     0x7f0a3848a000   0x1ff000    0x25000
```

```
     /lib/x86_64-linux-gnu/libtinfo.so.5.9
[...]
```

从地址空间的分配能看到，*0x400000-0x6e7000* 是第一段有效内存空间，低于这个范围的，都是无效的。上面%rsi 为 0x0，就明显是一个无效的地址空间。

在 ubuntu 上，与 core 相关的说明：https://wiki.ubuntu.com/Apport

3 环境配置

这些天有些烦闷，因为总觉得自己进度很慢，而且在做些重复的事情。这几天决定配一个自己的 docker 虚拟机，专门用来调试，决定把系统选在 ubuntu16.04，下面我会记录配置过程，当然，重点是对错误的记录。

下面来看第一个错误，是在我配置了 gef 之后，发现系统的字符集不足：

```
  ~ gdb a
...SNIP...
Reading symbols from a...done.
Python Exception <class 'UnicodeEncodeError'> 'ascii'
   codec can't encode character '\u27a4' in position 12:
   ordinal not in range(128):
gdb$

  /etc locale -a
C
C.UTF-8
POSIX
  ~ lolocale-gen en_US.UTF-8
  ~ lolocale-gen en_GB.UTF-8
  ~ update-locale
  ~ locale -a
C
C.UTF-8
POSIX
en_GB.utf8
en_US.utf8
```

完全摸不着头脑的错误，想想算了，遇到的错误越多，多 ubuntu 这个系

统结构的理解也越深入吧... 自己骗自己吧...

通过查资料，应该是缺乏了字符集： *en_GB.UTF-8 UTF-8*，既然找到问题了，就一点点解决吧...

4 幻灯片演示

5 从错误中救赎

6 问题探骊

这里是我在学习软件调试过程遇到的一些问题，有些解决了，有些没有解决，就全部记在这里，供学习完作为思考题。

6.1 动态链接库

1. 程序运行过程中，动态链接库是何时加载到内存空间中？
2. 动态链接库加载后，存储在内存空间中的哪里？
3.

6.2 Stack 相关

1. memcpy 引起栈溢出，为什么不会影响 memcpy 函数？
2. memcpy 函数为什么不需要对 rbp 进行保存?

6.3 Heap

1. malloc 申请了堆空间后，如何查看堆的位置？
2. 指针指向一个地址，那么结构体里面的函数代码地址怎么被确定的？