An Introduction to MOSDEF

Dave Aitel Immunity, Inc http://www.immunitysec.com





LLUSTRATE TRUE RISK





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 priviledged access
- _ VERY annoying to transfer files
- Cannot easily do portforwarding or other advanced requirments

_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 torwards 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 postexploitation 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

Exploited Process

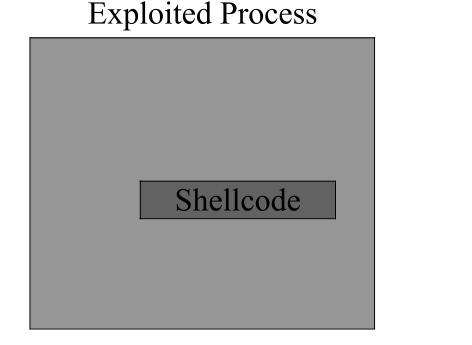
Shellcode

Attacker



_Shellcode Missions

- _ Step 1 is to establish back-connectivity
- _ Step 2 is to run a mission



Attacker



_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)
 Attacker

Exploited Process

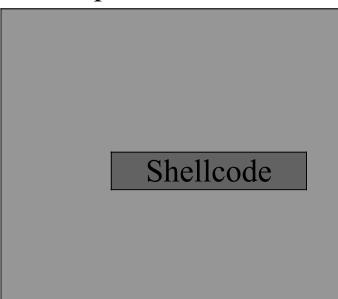
Shellcode	



_Running a Mission

- _ Step 2 is to run a mission
 - Recon
 - Trojan Install
 - Etc

Exploited Process



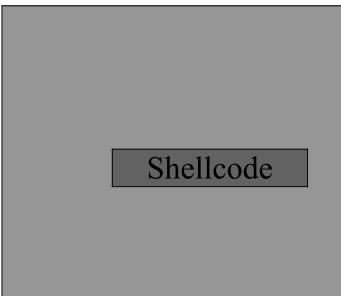




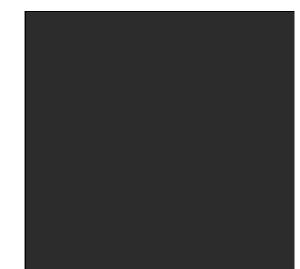
_Running a Mission

- Missions are supported by various services from the shellcode
 - Shell access
 - File transfer
 - Priviledge manipulation

Exploited Process







_Mission Support

- 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

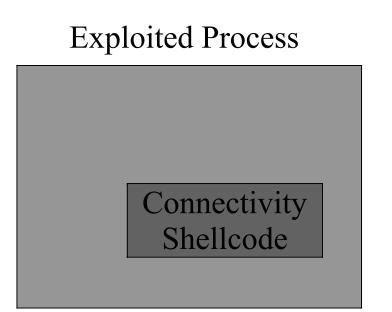
Attacker

Exploited Process	
Shellcode	



_Mission Split

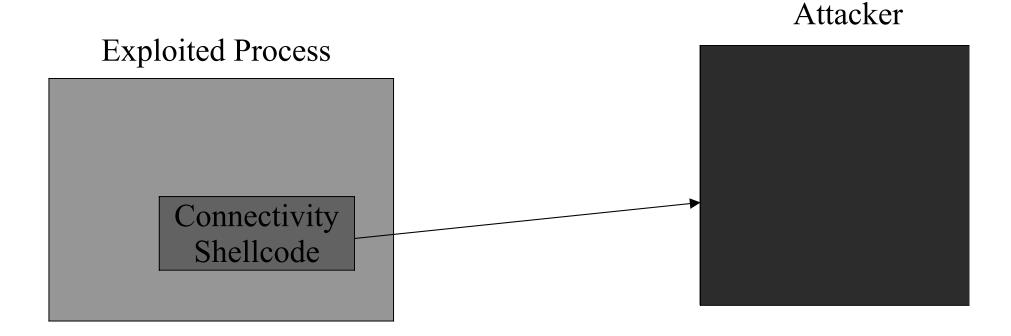
- Solution: split the mission from the stage1 shellcode
 - Smaller, more flexible shellcode



Attacker

_Mission Split

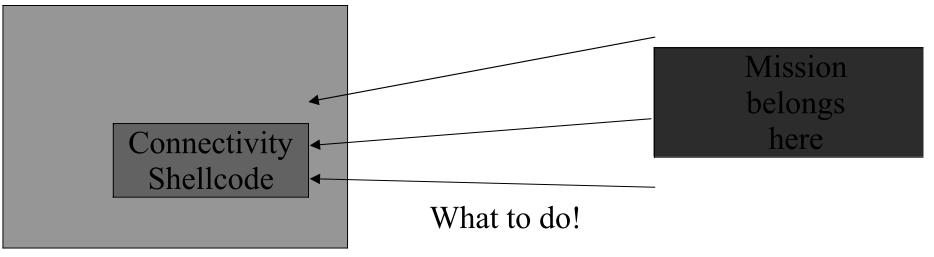
- Solution: split the mission from the stage1 shellcode
 - Smaller, more flexible shellcode
 - Simple paradigm: download more shellcode and execute it



_Stage 2

_ Options:

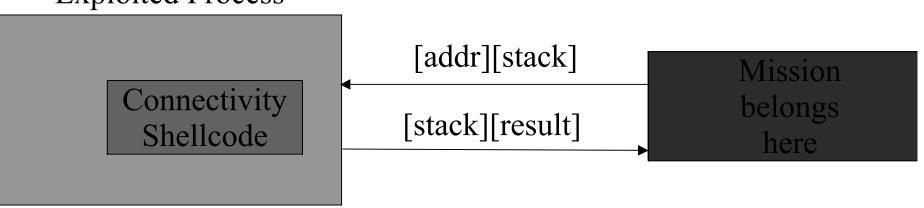
- Send traditional execve() shellcode
 - _ Or similar 1-shot mission shellcode
- Establish remote stack-swapping service
- Establish remote MOSDEF service
- Load and relocate an ELF/DLL Attacker
 Exploited Process



_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

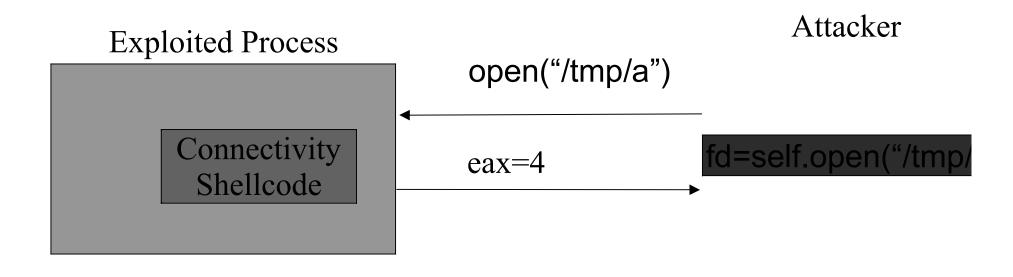


Attacker

_Stack Swapping - Benefits

Interactive with remote machine:

 Allows for interactive mission support on top of fairly simple shellcode



_Stack Swapping - Benefits

Most function arguments on Unix are easy to marshall and demarshall

```
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

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

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

- 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

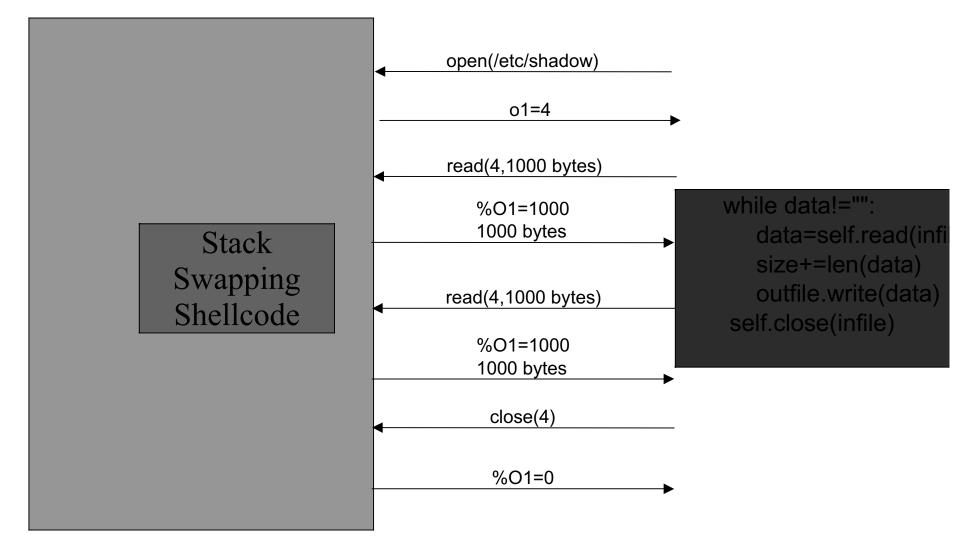
Basic stack swapping download code for Solaris

```
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()
```

File download protocol from randomhost.cn

Exploited Process

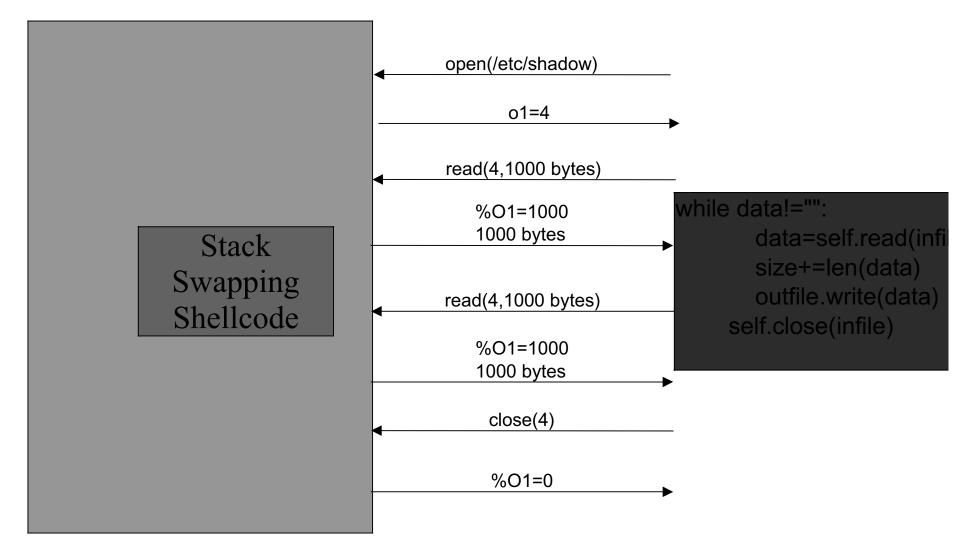
Attacker



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

Exploited Process

Attacker



- All iterative operations take 1second * 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!"

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 requirments

- _ 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

creat(self,filename):

.....

inputs: the filename to open

outpts: 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 Icreat function!" return -1

#ok, now we know the address of Icreat 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() raturn fd

MOSDEF sample

What does this take?

```
ef lcreat(self,filename):
```

fd=self.readint()

return fd

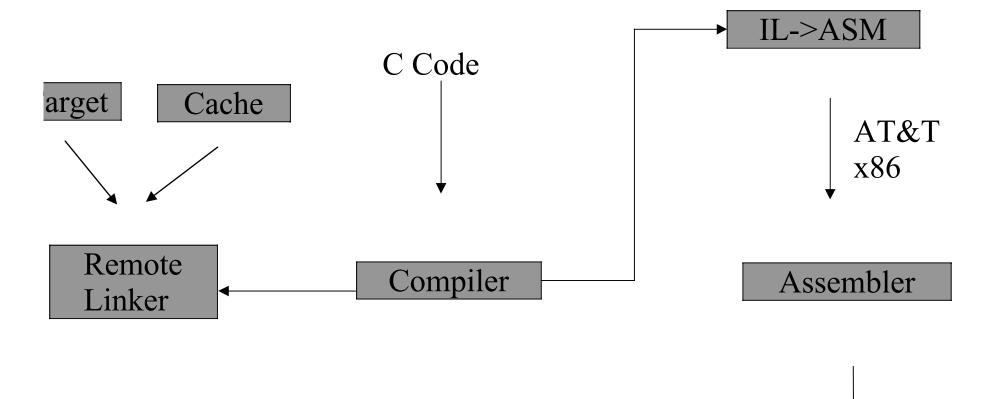
inputs: the filename to open outputs: returns -1 on failure, otherwise a file handle truncates the file if possible and it exists

```
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)
```

_A C compiler _An x86 assembler _A remote linker

_MOSDEF portability

_ Internal architecture



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\x15\x44\x55\x11" 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
 - Transfering 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 <u>http://felinemenace.org/~nd/</u>
 - 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/