KNOWING YOU'RE SECURE

# *Advanced Ordnance*

V 1.2

September 13, 2004

Dave Aitel
dave@immunitysec.com

# *Advanced Ordnance* Project Goals

- Examine and develop advanced replicating programs
- Examine and develop remote information retrieval techniques
- Examine how to properly use these techniques to perform advanced testing of network security

# MOSDEF

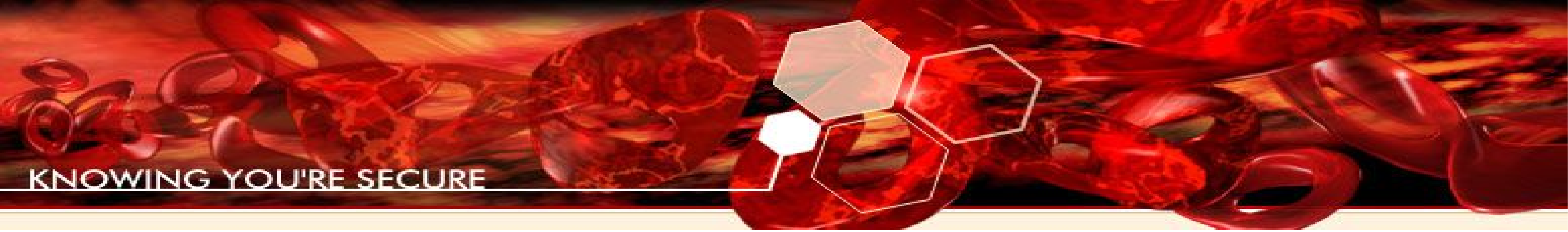
- MODSEF was first presented to the public at Blackhat Federal 2003, and is now usable
- It provides a pure Python compiler and assembler for use in CANVAS's exploitation engine
- MOSDEF is publicly available under the LGPL
  - http://www.immunitysec.com/

# Overview

- After you've overflowed a process you can compile programs to run inside that process and report back to you
- Goal: Support Immunity CANVAS
  - A sophisticated exploit development and demonstration tool
  - Supports every platform (potentially)
  - 100% pure Python

# MOSDEF Design

- 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



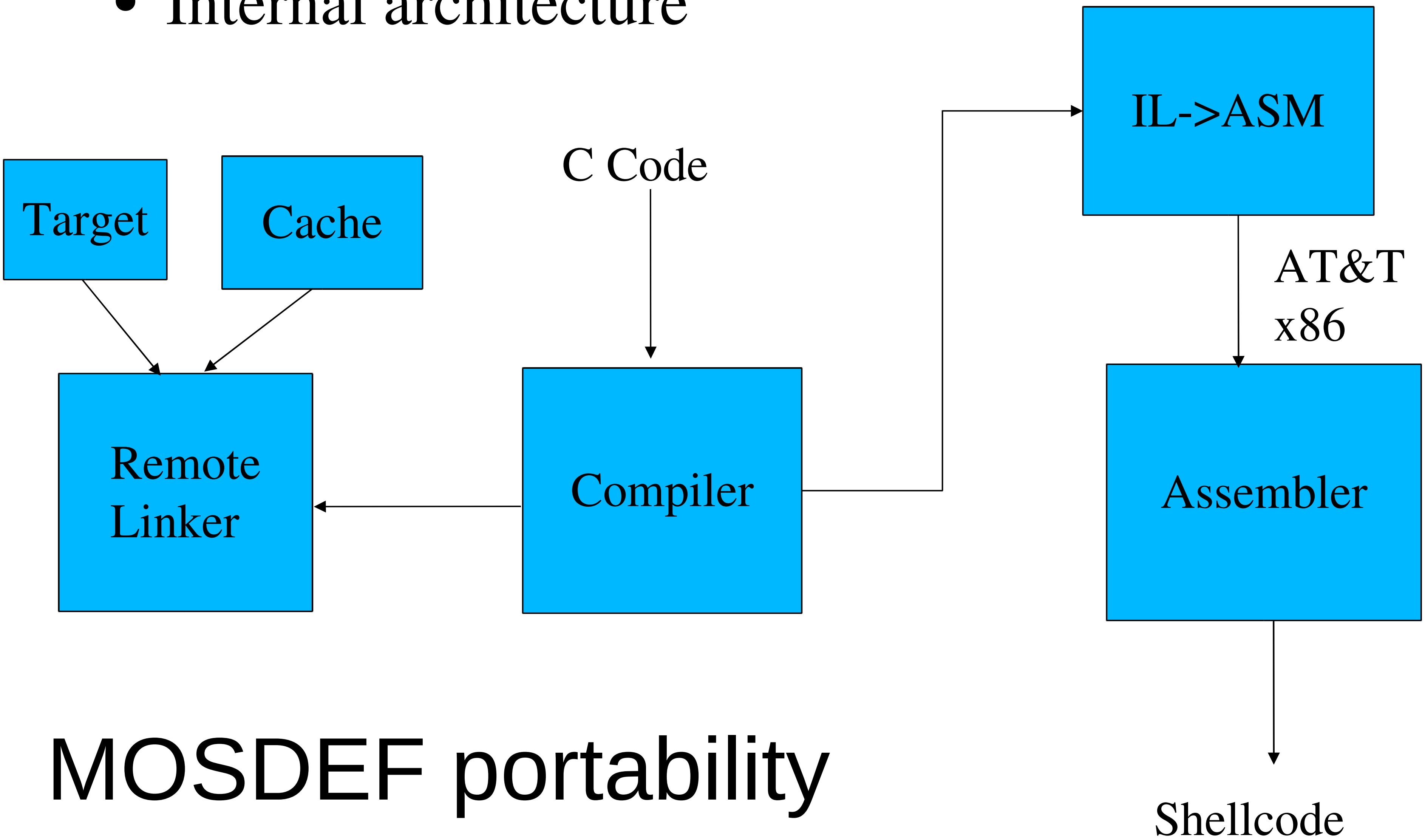

# MOSDEF sample

```
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
```

- Needed
  - A C compiler
  - An x86 assembler
  - A remote linker

- Internal architecture




# MOSDEF portability

# 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
  - 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, MOSDEF is licensed under the LGPL
- http://www.immunitysec.com/MOSDEF/

# Other Projects of Interest

- Hoon - http://felinemenace.org/~nd/
  - X86 AT&T assembler for shellcode written in Python
- Shellforge
  - A Python script to parse GCC generated .o files and generate shellcode
- The Grugq's userland-exec

# Advanced Ordnance

Taking MOSDEF one step further

# Why a worm?

- Self-replicating programs manage to surprise people with where they get. We need to capture that serendipity
- Worms are the ideal platform for distributed algorithms
- We may only have hours of target-window
  - May have a certainty of being discovered
  - May be losing a 0-day
  - May be losing connectivity
  - May be able to come back later, but various intermediate hosts are different

# Mental Position

- Networks are still mostly flat
- Mission oriented: Given X I should be able to Y
- Manual (operator-dependent) network penetration is hard to scale in both speed and size
- Not all hosts are on the network at once
- **Host to host jumping is a concept that breaks down in the time domain**

# Given

- Access to one host internally
- Ability to create reliable exploits

KNOWING YOU'RE SECURE

# I should be able to:

- Get every file on the network named “*.xls” with “Salary” in it
- Be able to tune to multiple levels of covertness
- Be able to control the replicating program and restrict it to a certain level of hosts and networks
- Prevent forensic analysis from knowing what files I recovered, if any
- Prevent automated response and analysis of my replicating program, defeating IDS if necessary

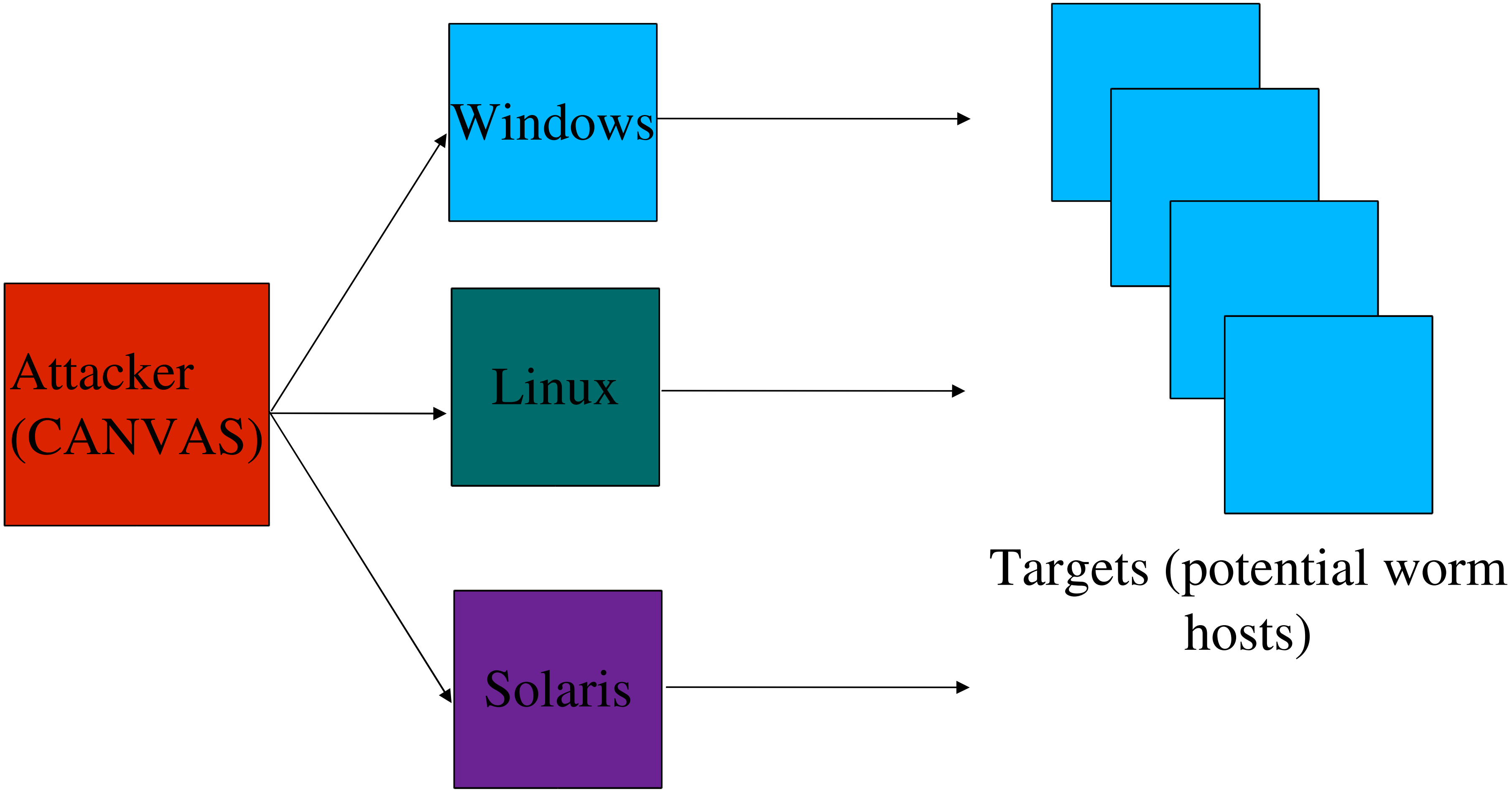
# Agenda

- Three stages
  - Injection method
  - Payload creation
  - Strategic denial and deception

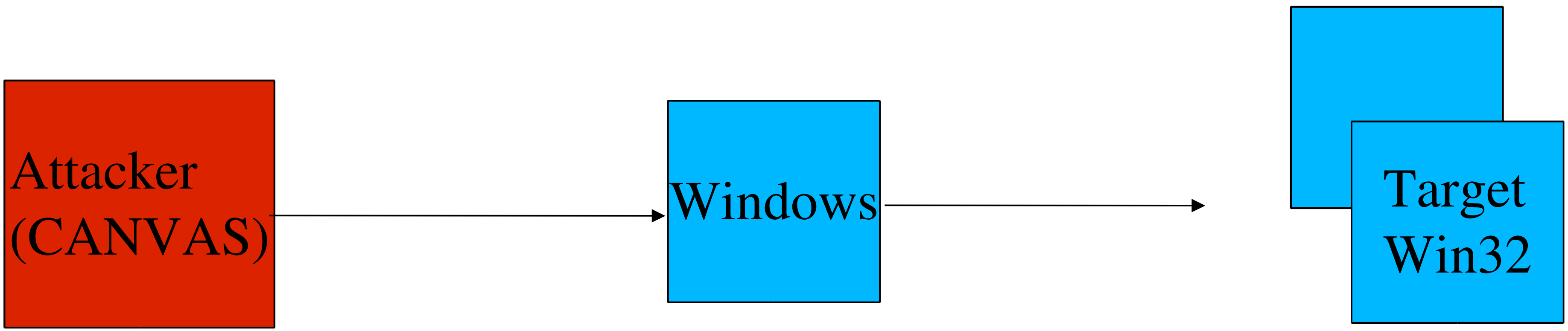
# Injection Method

- Easiest to assume that we are in control of a process from some given exploit
  - Should not have to be the same exploit as the worm will use
  - Interesting case is to assume we have the same OS and architecture
    - And program

# Injection Possibilities

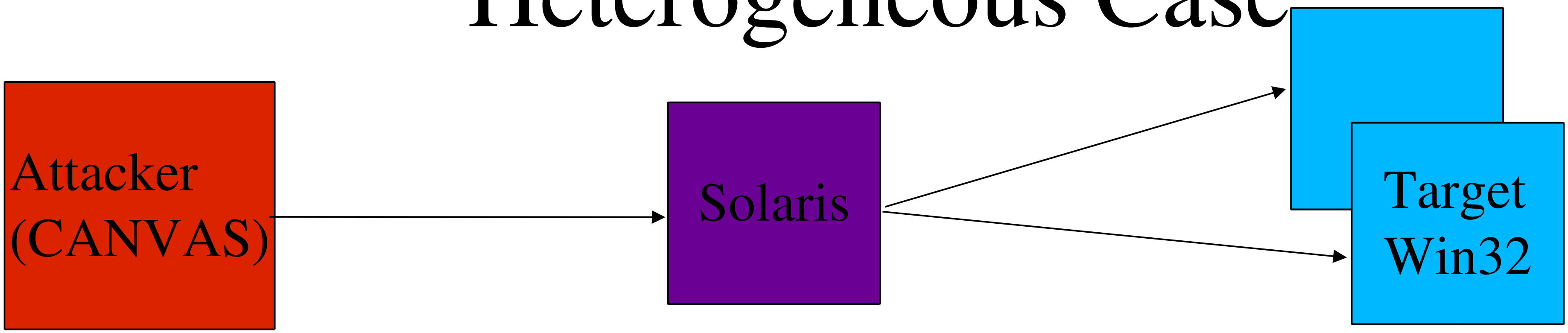


# Homogeneous Case




- Same platform allows us to infect current process by emulating target environment
  - Drawback: forensic analysis of current host will reveal initial penetration method (which may be a different process than the worm's target)
  - Can disconnect after infection to allow current host to perform further infections autonomously

# Heterogeneous Case




- Must emulate worm behavior to infect initial hosts
  - Worm behavior does not have to be a simple send()!
- Must scan for target set
  - Or have pre-placed target list
- Create custom payload
- Infect targets

# A Worm Creation API

- Worms have a static environment, based on target assumptions
- Worms must be self-reliant
- Worms must be able to have a complex exploitation procedure
- A good engine should support mutation

# Example

- Start of shellcode has happened
  - Must select and infect targets
    - Create payload from current mangled shellcode
    - Perform scanning
    - Perform infection
  - Must perform payload operations
    - Exfiltration, destruction, etc
  - Space is a premium, since you carry your house with you like a turtle

KNOWING YOU'RE SECURE

# ShellcodeGenerator

- The problem is creating a shellcode which can replicate
- We use MOSDEF as an integrated compiler/assembler
  - Treat all aspects of worm creation as defining a special-purpose compiler
  - We operate on the string that is the assembly language to create a payload
- Similar to current CANVAS win32 shellcode generation
- Can also use advanced MOSDEF C compilation at times

# Worms are special cases of shellcode

- You may not want to decode your worm at all!
- You may need to decode after performing special operations (such as copying) first
- Your payload may need to define a complex network protocol
- Your worm may mutate as it goes along to hold state (TTL, etc)

# Simple example

- Using testvuln1.c as a target on the Windows operating system
  - Can assume esp points to current shellcode
  - Psuedo-code of worm:
    - Copy current payload at esp to another location and store that location
    - Decode shellcode
    - Generate an import table with connect(), send(), random number generator function
    - For each random IP, connect(); send(sizeof(payload)); send (payload);

# Building a payload

- testvuln1.c required minimal protocol creation, since it just needed to be sent a size and then the shellcode to be executed
  - Size should be little endian
  - Unlimited size
  - No filter

```
sc=advancedordinance_x86()
sc.TTL(5)  #network hops to go through
sc.maxSize(0xffffffff) #compiling routines take this into account
sc.targetsRange(“192.168.1.0/24”) #changes our randomization function
sc.setTargetPort(5000) #used for tcpconnect
sc.setReplicateLoops(500)
sc.setStyle(“payloadfirst”) #run the payload, then replicate
sc.setSelfPtr(“esp”) #the pointer to use at our start
sc.code=”””
  copyself stackalloc  #set “selfloc” variable to point to new copy
  decTTL  #when we reach 0 we skip the  replicating block
  replicatestart #a label for control flow
    getrandomtarget
    tcpconnect #open a socket to the target
    sendint 2000 #our size plus some
    sendmem selfloc #send the worm to the next host
    tcpclose #close the socket
  replicateend #loop if necessary
  payloadstart #a label for control flow
```

# BlackIce ICQ overflow (Witty)

- Requires protocol header – uses offset from edi to locate it
- Written entirely in non-zeros, so doesn't need encoder/decoder and can use itself as payload
- Covertness:
  - Chooses random destination ports and from addresses

```
sc=advancedordinance_x86()
sc.setFilter("\x00")
sc.TTL(5)  #network hops to go through
sc.maxSize(5000) #compiling routines take this into account
sc.targetsRange("192.168.1.0/24") #changes our randomization function
sc.setReplicateLoops(500)
sc.setStyle("payloadfirst") #run the payload, then replicate
sc.setSelfPtr("*(edi-8)") #the pointer to use at our start
sc.code=""""
  getself  #set "selfloc" variable to point to new copy – we don't have a decoder loop
  decTTL  #when we reach 0 we skip the  replicating block
  replicatestart #a label for control flow
    getrandomtarget
    udpconnectrandom 4000#open a socket to the target from port 4000
    sendmem selfloc #send the worm to the next host
    udpclose #close the socket
  replicateend #loop if necessary
  payloadstart #a label for control flow
    #nothing yet.
  payloadend""""")
```

# Notes on Payload building

- Handling corruption
  - Building a nop bridge
  - Searching for clean copies of payload in memory
- Shortcuts
  - Delaying (or not doing) decoding
    - Special purpose compiler helps us write filter-passing shellcode
  - Delaying function table building
    - Using existing import tables where necessary

# Why use MOSDEF

- Having control of the entire compilation chain allows for us to do fairly cool things such as:
  - Create many versions of the same worm
  - Write our worm in C, and have it automatically mutated to avoid bad characters with certain filters
  - Dynamically and programatically update our worm to account for the conditions of the target network
    - Feeds into our need to automate entire process with AI

# Language Creation Goals

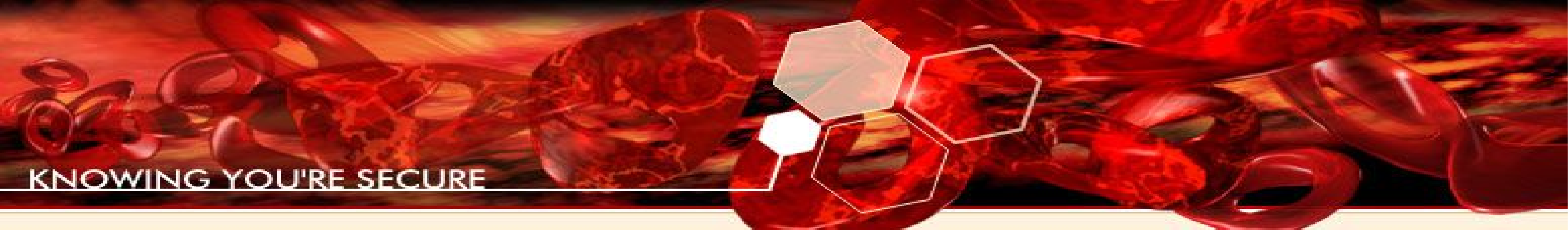
- Worm payloads are special purpose shellcodes, which are already handled by MOSDEF's shellcode language compilers
- Goals of worm language
  - Allow creation of tight shellcode
  - Allow quick and easy modification both manually and programatically
  - Allow extensive flexibility to add new functionality
    - Function pointer tables are automatically generated, etc
  - Platform independence where possible
  - “Make it easy to convert an exploit to a worm”

# Additional Notes

- Porting an exploit to a self replicating program can be assisted by automated vulnerability analysis tools
  - "Exploit generators" such as "autosploit.py" which examine the environment of a vulnerability and attempt to match exploit code to it by way of exploit "fingerprint" generation
- You may pay a reliability price for having to describe a vulnerability in the terms of a worm

# Strategic Denial and Deception

- Mutating worms
- Hidden initial infection methods
- Preselected targets
- Payloads and exfiltration
- Restrictions on host range and TTL




# Potential Detection Methods

- Blackice or other host IDS
- Network traffic surges
- Lucky forensics experts

# Blackice/HIDS can be disabled

- However, it is difficult to do this quietly and in only a few bytes of shellcode
- Many HIDS configurations exist
  - Because we create worms dynamically, we can also create worms specific to our host network!
- Pays to be prepared with special purpose shellcodes for your target's network
  - What software do they require all users to install?
- HIDS are still rare

# Network surges

- Speed/Surge trade-off
- Distributed file transfers of large amounts of data
  - Special purpose network protocols
- Can piggyback network's existing protocols (SMB, etc)
  - Have to avoid CheckPoint “ApplicationIntelligence”, etc

# Surgeless protocols

- Disk space is cheap, covert storage is almost cheap
- Problem: Transfer all interesting files in a protected area that has been pierced by our worm back to our host, without knowing where our host is
- Solution: Send every interesting file to every host

# Worm File Transfer Protocol 1

- Assuming a small set of files is interesting (has keyword and is a spreadsheet)
    - Files may exist on more than one host
  - Transfer all files to every host I can talk to
    - Can use simple size+data network transfer to copy
    - File size and filename make good pseudo-hashes
    - Ask permission first – drop connection if file already exists or disk is almost full
- Eventually all the files that the worm was able to reach will exist on all the hosts the worm was able to reach

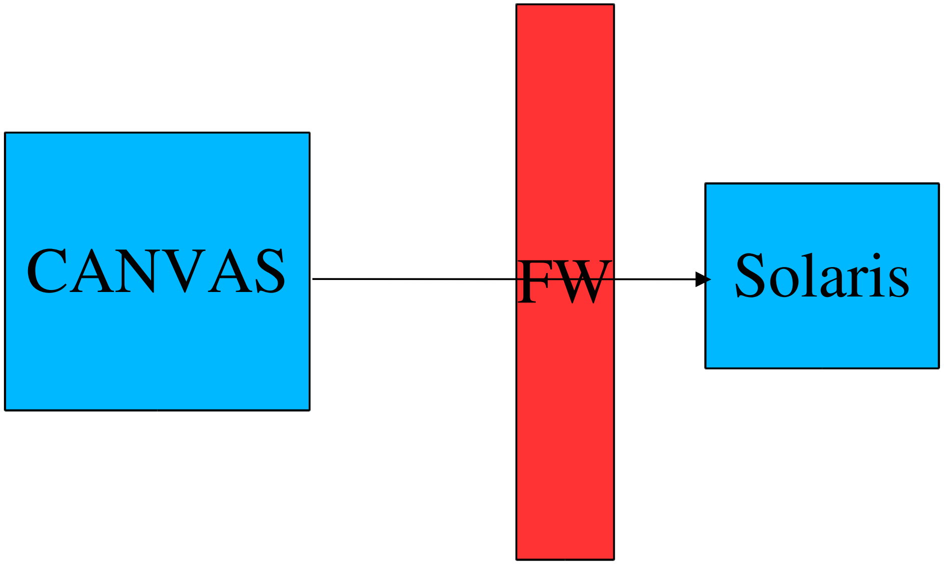
# Other obvious file transfer options

- Exfil directly to an outside host:port via HTTP or other protocol
  - Can use network's proxies via autodiscovery
- Use email
- Etc
- All this is easier to build with MOSDEF than by hand!

# Defeating forensics

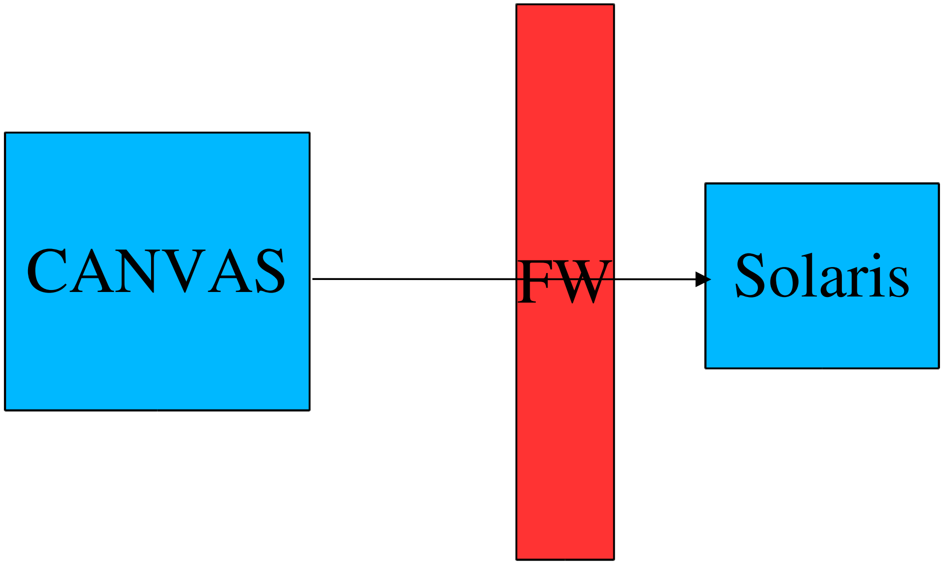
- Mutate the worm to erase one of the payloads after MaxTTL/2
  - First phase worm will install backdoors or accomplish mission
  - Second phase worm will be analyzed by target's forensics team and declared mostly harmless

MOSDEF
link with target
through Firewall




- Penetrates initial web server in DMZ
- Automatically conducts recon and notices that CA Unicenter is running (and we have 0day prepared)
- Decides, based on covertness level and “AI's” various decision making alg's to launch a worm to retrieve documents labeled “Top Secret”

MOSDEF
link with target
through Firewall




- Uses arp -a cache and other methods to fill a target list with vulnerable targets
- Generates worm payload using AO
- Uses MOSDEF/Other remote execution engine to emulate a worm and infects all targets found
- Also uses MOSDEF/etc to emulate the worm's transfer protocol to collect files
- Cleans up Solaris box, and leaves when done

# Conclusion

- Worms can be useful tools and should be exploited to provide a reaction capability that outstrips a network defender's ability to adjust and compensate
- You can only model what you understand – we hope to get some benefit out of network effects we don't understand yet
- Automated defenses require automated attack platforms
- Multi-exploit and multi-platform worms are even more useful and may require their own special purpose languages – subclassed from AO itself

# Acknowledgments


- Authors of worms everywhere
- Homies in Iraq (ph00dy, et. al.)
- Justine Aitel
- #convers, esp Oded for reliable heap overflow discussions
- Halvar Flake