

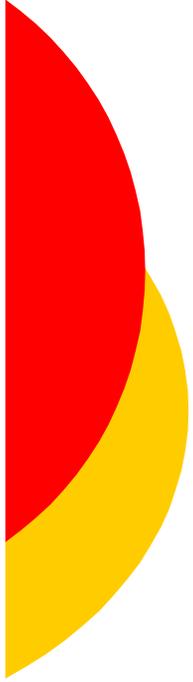


# Runtime Decompilation

---

The 'GreyBox' process for  
Exploiting Software

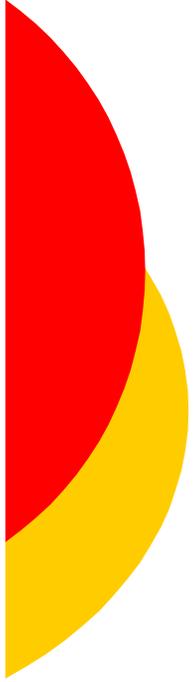
By: Greg Hoglund  
hoglund@hbgary.com



# Introduction

---

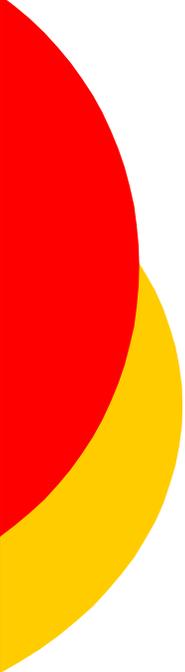
- This presentation is an overview of the lab process used by HBGary™ to locate and exploit software bugs
- The value is reduced time investment and formalizing a 'black art'



# The Deconstructionist

---

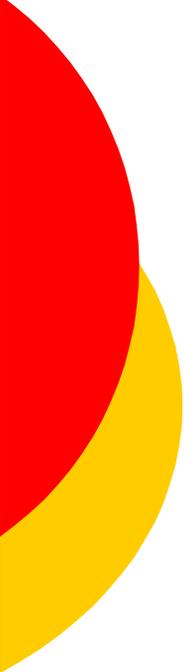
- Taking a system apart is about uncovering mysteries
- Having secret knowledge attracts the human psyche at the deepest level
- There is an age old battle between those that create systems and those who take them apart
  - i.e, cryptoanalysis



# Why Exploit Software?

---

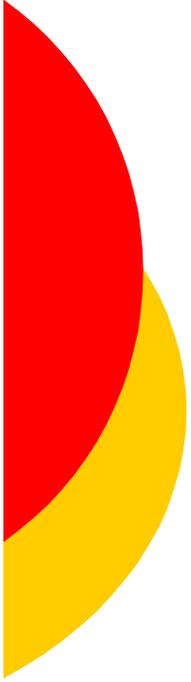
- Exploits are worth money
  - A vulnerability can be worth over \$100K
    - The vendor costs are huge for a public vulnerability
  - An exploit costs less than physical 'bugs'
- Exploits are worth lives
  - An exploit is safer than physical penetration
- Exploits are strategic
  - Disable or control the information systems of your enemies decision cycle



# Survivability

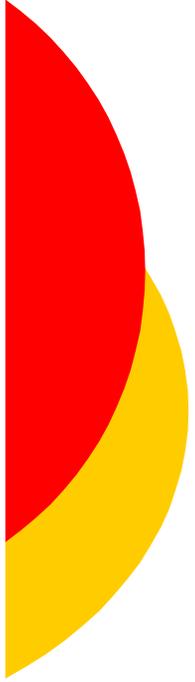
---

- Exploits have a lifetime
- Every use of an exploit has the potential to compromise the asset
- Exploits depend on bugs and your enemy may also find the same bug
- The public may find the same bug
- Once public, many exploits can be protected against or detected via an IDS



---

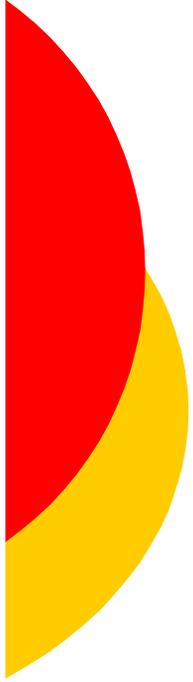
In order to maintain a battle advantage, your offensive information capability must include a lab process for finding and exploiting new software bugs



# ***Chapter One***

---

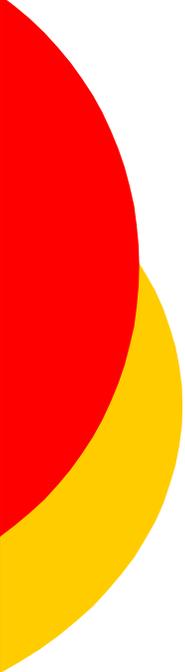
The Bugs!



# The Bugs

---

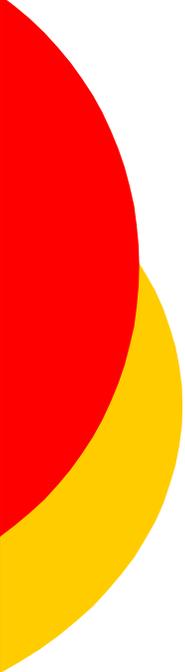
1. Buffer Overflows
  - a. Lack of bounds checking
  - b. Arithmetic errors
2. Parsing Problems
  - a. Input filters and normalization
3. General State Corruption
4. Race Conditions



# Buffer Overflows

---

- Old News, but still most common today
  - Because of speed, most server software is still developed in c/c++
- Will remain common until old compiler technologies are abandoned
  - Strongly typed languages, such as C#, eliminate simple string overflows



# Parsing Problems

---

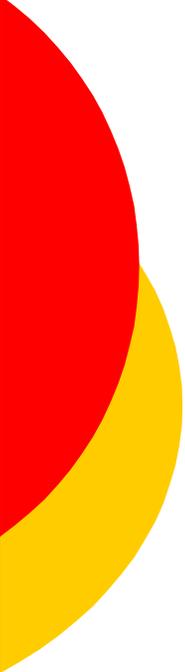
- Not solved by better compilers
- Solved only by good algorithms
- To eliminate parsing problems requires standardized algorithms
  - Similar to peer review on crypto systems
- This will never happen
  - Parsing problems here to stay



# General State Exploits

---

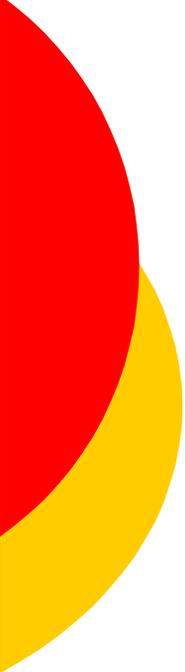
- States control decisions
- Users can cause state transitions
- Some states are insecure by nature
- State exploits are found by sending commands in the exact order and context to arrive at the insecure state
- Only solved by *provably* correct systems
  - Humans are never going to build provably correct systems



# Race Conditions

---

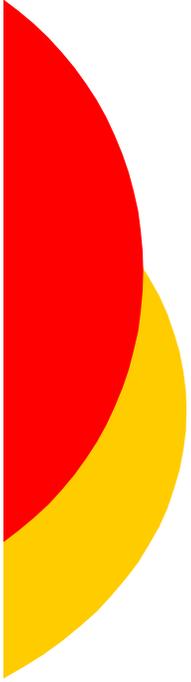
- State problems are going to be difficult to measure and control
- When state is managed over many nodes, the problem becomes even harder
  - When state must be synchronized among nodes, we have race conditions
  - The problem is compounded greatly
  - This is the 'buffer overflow' of the future



# *Chapter Two*

---

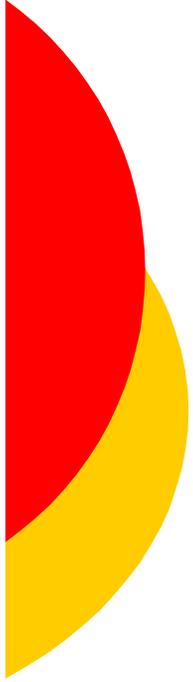
What is the  
'GreyBox' process?



---

## **GreyBox:**

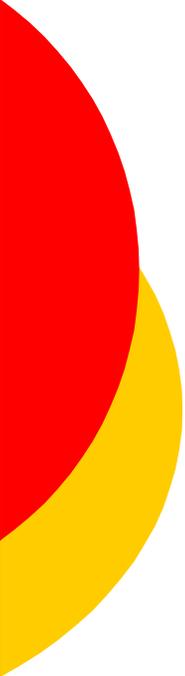
Combining both static analysis and runtime fault injection to maximize coverage of a software programs' state-space. Typically used to detect and isolate fault states in a software system.



# White Box

---

- In theory, operating with full knowledge about the inner workings of the system
  - At best, we only have an approximate understanding of the builder's intent
- White box analysis involves "deadlistings" static disassemblies of the binary
  - Source code is an added advantage, like having really good documentation for the deadlisting
- The software is ***not*** being executed



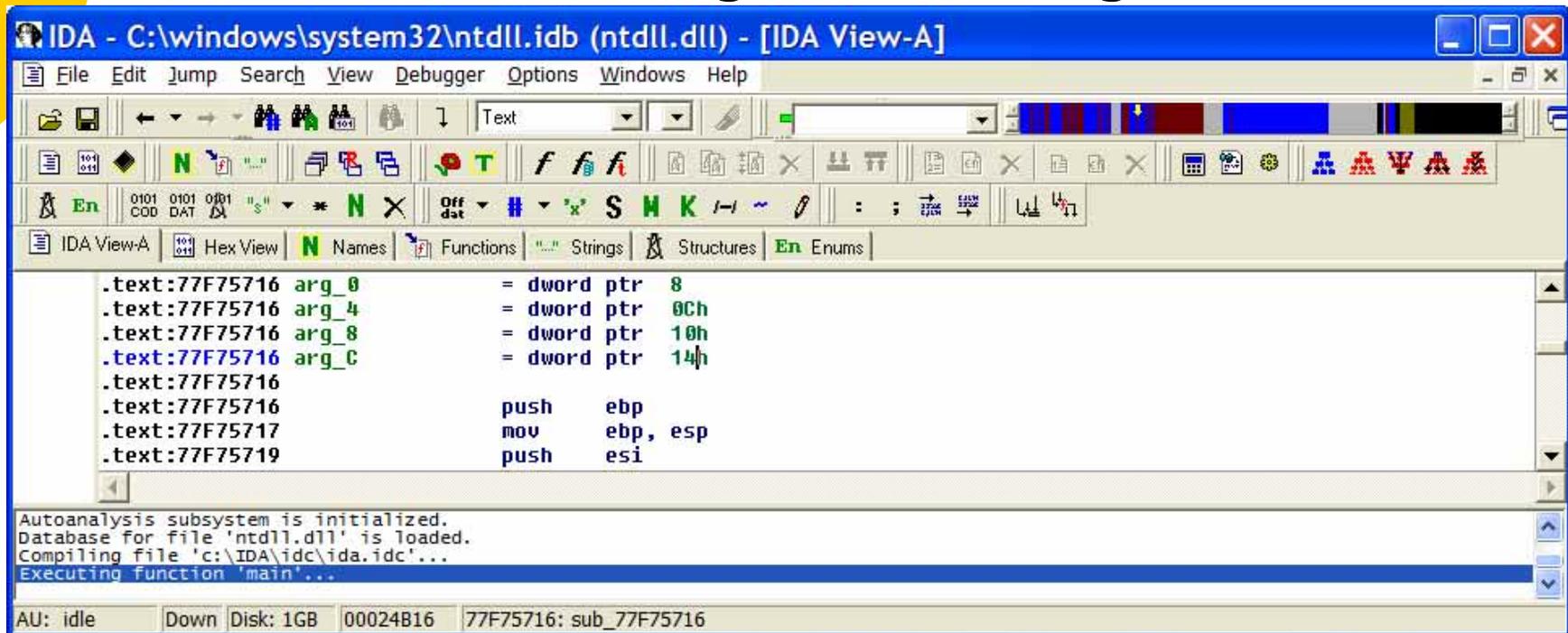
# First Pass

---

- When confronted with a new binary, the HBGary team fires the binary through BugScan™
- We obtain a report within minutes to assess whether the programmers use secure coding practices
  - Typically they do not
- We use the BugScan report to prioritize which binaries will be analyzed first
  - Binaries with bad reports are hit first

# Manage the Deadlisting

- IDA-Pro allows you to manage and comment a large deadlisting

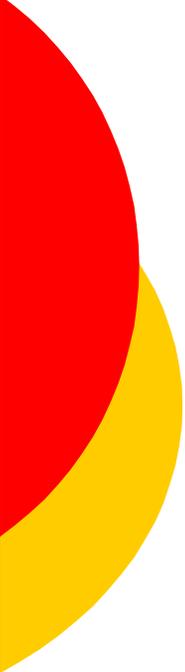


The screenshot shows the IDA Pro interface for the file 'C:\windows\system32\ntdll.idb (ntdll.dll)'. The main window displays assembly code in the 'Names' view. The code includes several instructions with comments and parameters:

```
.text:77F75716 arg_0 = dword ptr 8
.text:77F75716 arg_4 = dword ptr 0Ch
.text:77F75716 arg_8 = dword ptr 10h
.text:77F75716 arg_C = dword ptr 14h
.text:77F75716
.text:77F75716 push ebp
.text:77F75717 mov ebp, esp
.text:77F75719 push esi
```

The status bar at the bottom indicates the current state: 'AU: idle', 'Down', 'Disk: 1GB', '00024B16', and '77F75716: sub\_77F75716'. The console window at the bottom shows the following output:

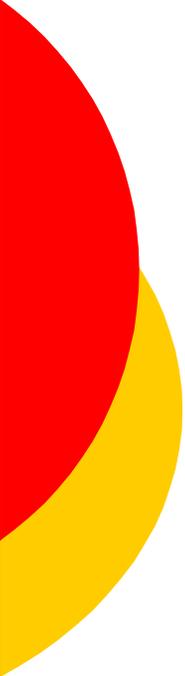
```
Autoanalysis subsystem is initialized.
Database for file 'ntdll.dll' is loaded.
Compiling file 'c:\IDA\idc\ida.idc'...
Executing function 'main'...
```



## Is it actually exploitable?

---

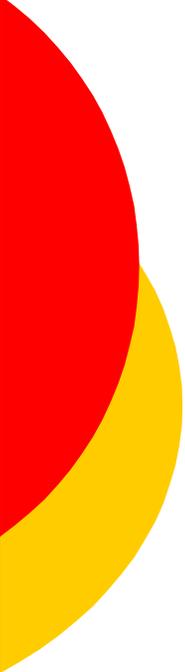
- Depends on many variables in the environment
- **All** automatic analysis tools have this problem
- It almost ***always takes an expert*** reverse engineer to determine if a condition is exploitable



## Does it matter?

---

- Even if a vulnerability cannot be reached **today** – what can you say about **tomorrow**?
- What if interface changes?
- What if code gets used from other locations?
- Is the original author going to be maintaining this code in 10 years?



# Automatic Bug Detection

---

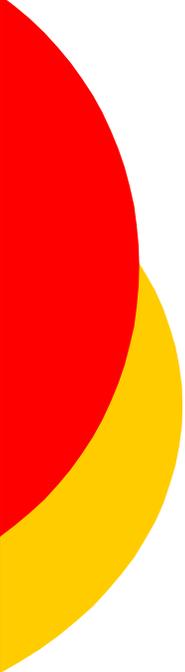
- The bug must have a defined pattern, it must be ***schematic*** in nature
- Effective when certain conditions exist
  - Availability of type information
  - Separation of data and code
  - All instructions can be recovered
  - Data that drives control flow can be mapped



## Branching Decisions

---

- Many branches are made based on values that are calculated at runtime
- The static analyzer must emulate execution to determine these values
- At some point, the emulation becomes computationally equivalent to running the program in the first place. How much emulation is enough?



## Backtraces reach dead ends

---

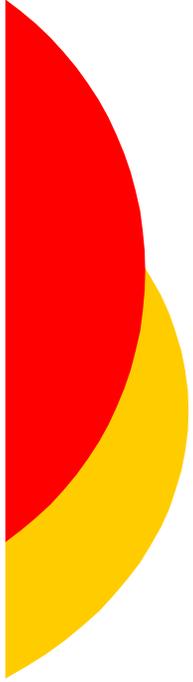
- We backtrace up to 64 steps from a vulnerable function
  - Every branch is exercised
- Back traced cross references can be used to connect input with a code location
  - For example, does a previous function take input from the network?
- Many times a static backtrace dead-ends
  - Windows message handler



# Black Box

---

- All we see are the outputs from the software – no inner workings
- Requires deep protocol knowledge
- 'Fuzzers':
  - Hailstorm and Spike



## Black Box is not stand alone

---

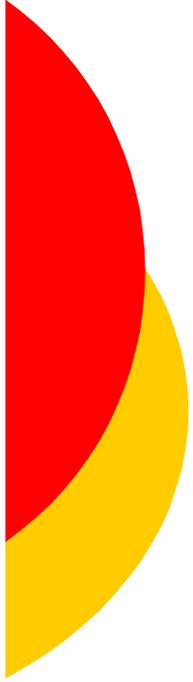
- Black box testers take FOREVER to complete their input sequences.
- If the program is slow, this compounds the problem
- Amounts to 'brute forcing'
- Finding bugs with pure brute force is mostly luck



## Blackbox State

---

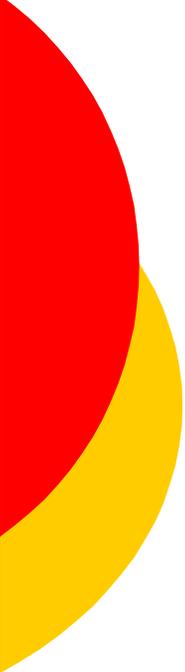
- Typical network software is highly stateful
- A client must be able to maintain a complex state in order to communicate effectively with the target
- Modeling highly stateful clients from scratch is very time consuming and prone to error



## Instrumented Clients

---

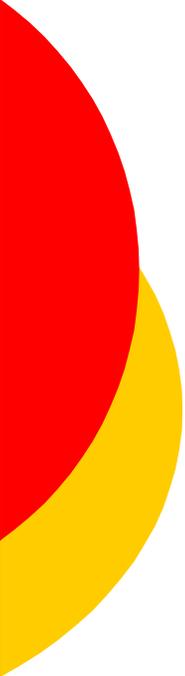
- Using a real client program eliminates most of the state issues
- You don't need to rebuild the wheel
- Fault injection is inserted ***in-transit*** by modifying the code within the client program
- The client program becomes a ***hostile mutant***



# Fault Injection Clients

---

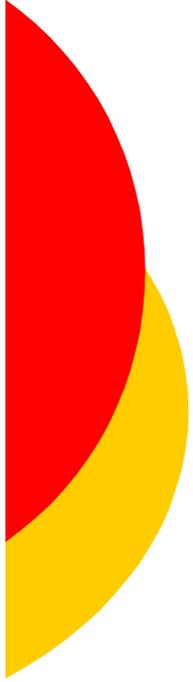
- If the protocol is proprietary you have two choices
  - Modify in the middle the packets
    - Only works if protocol is not overly complex and not encrypted – beware of authentication/encryption
  - Instrument proprietary client
    - Requires difficult call-hooking, time consuming



# Hooking clients

---

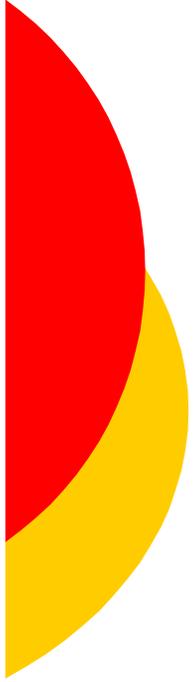
- Find location where pointer is held in register
- Put breakpoint on this location and modify the given string in memory
  - Cannot BO string w/o corruption
- Or, replace the pointer with another pointer
  - May cause state problems in some clients
- Using 'debugger' technology makes this whole process easier – no EXE patching required



## GreyBox

---

- Combine black-box injection with code analysis
- If you use a program debugger, your performing grey-box analysis
- Performed at runtime so software can be observed
- All instructions which are executed can be obtained. All data involved at these points can be tracked



## Interactive

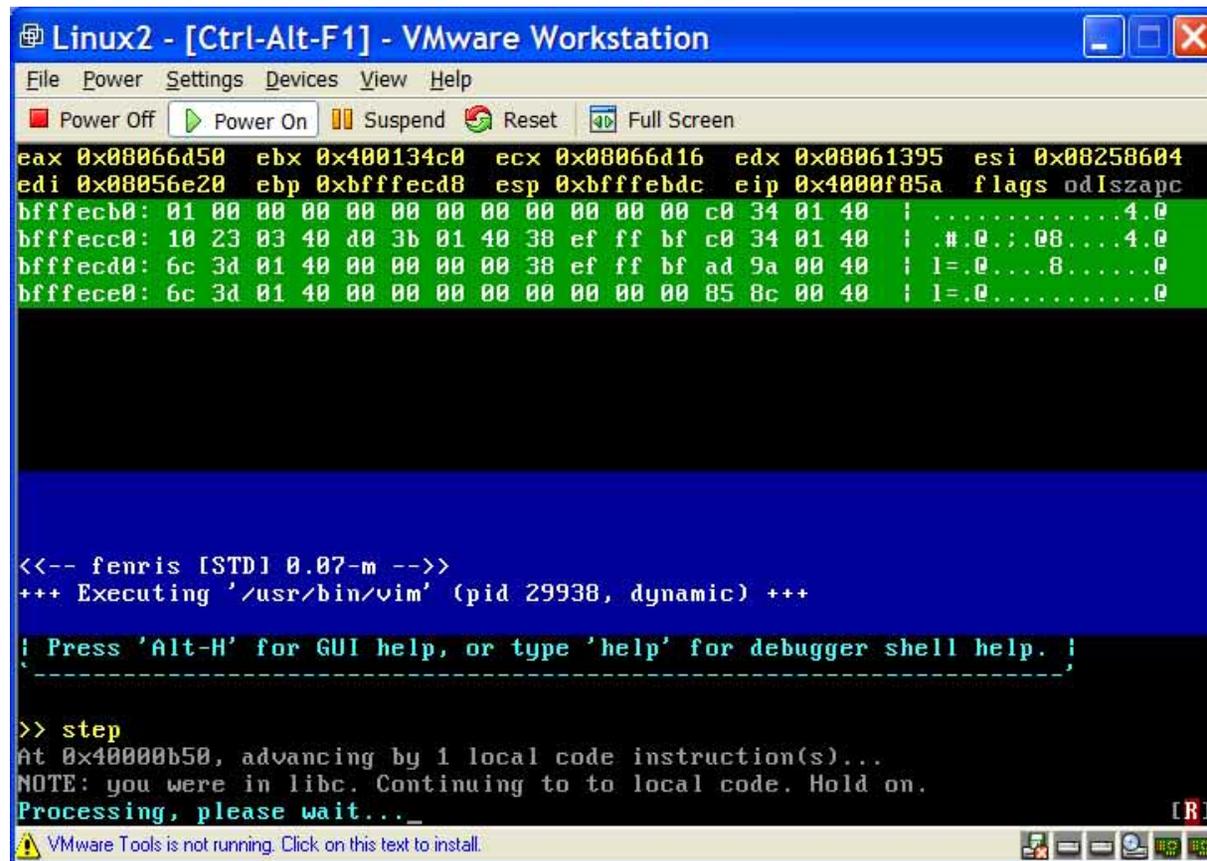
---

- Grey-box testing is an interactive process between a skilled engineer and the target program
- Tools used include SoftIce, OllyDbg, Aegir, Fenris, GDB, Tempest, and the MS-Visual C debugger\*

\*IDA-Pro has an integrated debugger, but the current version is not evolved enough for industrial level work

# Fenris

[www.bindview.com](http://www.bindview.com)

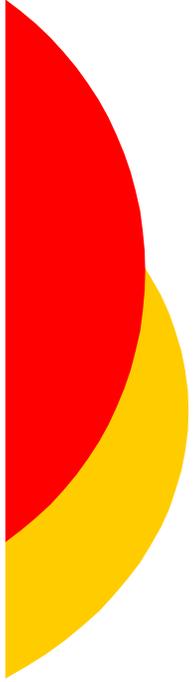


```
Linux2 - [Ctrl-Alt-F1] - VMware Workstation
File Power Settings Devices View Help
Power Off Power On Suspend Reset Full Screen
eax 0x00066d50 ebx 0x400134c0 ecx 0x00066d16 edx 0x00061395 esi 0x00258604
edi 0x00056e20 ebp 0xbfffecdb esp 0xbfffebdc eip 0x4000f85a flags odIszpc
bfffecb0: 01 00 00 00 00 00 00 00 00 00 00 c0 34 01 40 | .....4.0
bfffecc0: 10 23 03 40 d0 3b 01 40 38 ef ff bf c0 34 01 40 | .#.0.:08...4.0
bfffecd0: 6c 3d 01 40 00 00 00 00 38 ef ff bf ad 9a 00 40 | 1=.0...8.....0
bfffece0: 6c 3d 01 40 00 00 00 00 00 00 00 00 85 8c 00 40 | 1=.0.....0

<<-- fenris [STD] 0.07-m -->
+++ Executing '/usr/bin/vim' (pid 29938, dynamic) +++

! Press 'Alt-H' for GUI help, or type 'help' for debugger shell help. !
-----

>> step
At 0x40000b50, advancing by 1 local code instruction(s)...
NOTE: you were in libc. Continuing to to local code. Hold on.
Processing, please wait..._ [R]
VMware Tools is not running. Click on this text to install.
```



# ***Chapter Three***

---

How to use  
Bug Scan<sup>TM</sup>



## Easy Stuff – Introducing BugScan!

---

- BugScan is **extremely simple** to use
- Submit binary and get report
- Report cannot verify if conditions are actually exploitable
  - But it takes 30 seconds, not 30 hours
  - Defensive stance – don't wait for someone to attack before you protect yourself

# Submit a File



# View the Report

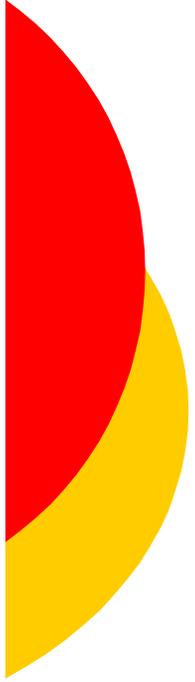
BugScan v1.0 [Browse Reports](#) [Analyze a File](#) [Log Off](#)

< Previous [Next](#) >

ID	Sev	Name	Risk	Description
<a href="#">0</a>	2	wsprintfA	overflow	Replace this call with the more secure call, wsnprintf. wsnprintf is a variant of wsprintf where the user explicitly specifies the length of the destination buffer. This feature helps avoid the possibility of the destination buffer being written past. Though not officially a part of the ISO C99 standard, this call is available in most modern compilers. Two problems can still persist: format string bugs and specifying an incorrect length for the destination buffer.
<a href="#">1</a>	2	lstrcpyA	low	Double check that the destination buffer is larger than the specified length in this call. If the supplied string is exactly equal to the size of the destination buffer, the string will not be NULL terminated.
<a href="#">2</a>	2	lstrcpyA	overflow	Replace this call with the more secure call, lstrcpy. lstrcpy is a variant of lstrcpyA where the user explicitly specifies the length of the destination buffer. This feature helps avoid the possibility of the destination buffer being written past. Though not officially a part of the ISO C99 standard, this call is available in most modern

Discussions [Subscribe...](#) Discussions not available on <http://216.27.185.83/>

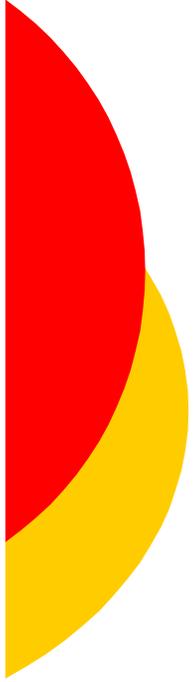
Internet



# Latest BugScan Reports from the Field

---

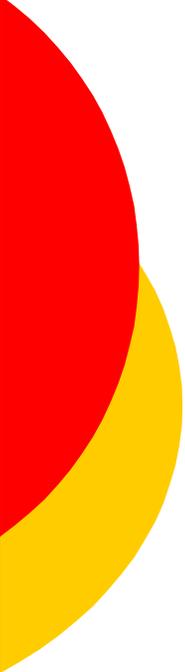
TO BE REVEALED AT CONFERENCE



# FREE BUGSCAN for BLACKHAT

---

- Use this logon to scan any binary, free for blackhat attendees for the next 60 days
  - [HTTP://www.hbgary.com/freeblackhat](http://www.hbgary.com/freeblackhat)



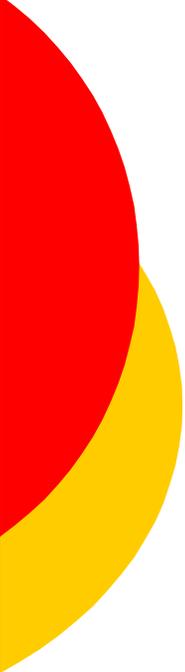
# *Chapter Four*

---

How

HBGary

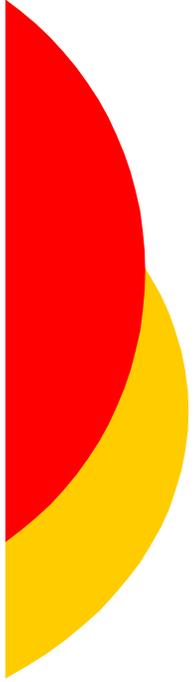
Uses GreyBox



# Hard Stuff

---

- Designed for experts
  - Not a product!
- Requires reverse engineering skills not limited to:
  - Runtime debugging
  - Assembly code
  - Protocols
  - Technical knowledge of programming bugs

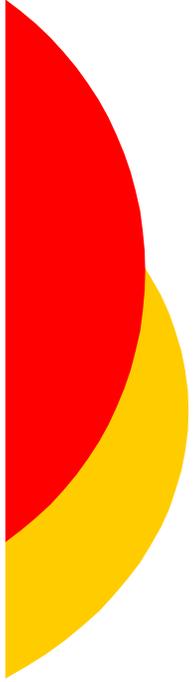


# Introducing TEMPEST

---



Free technology available for  
download from  
**[www.hbgary.com](http://www.hbgary.com)**

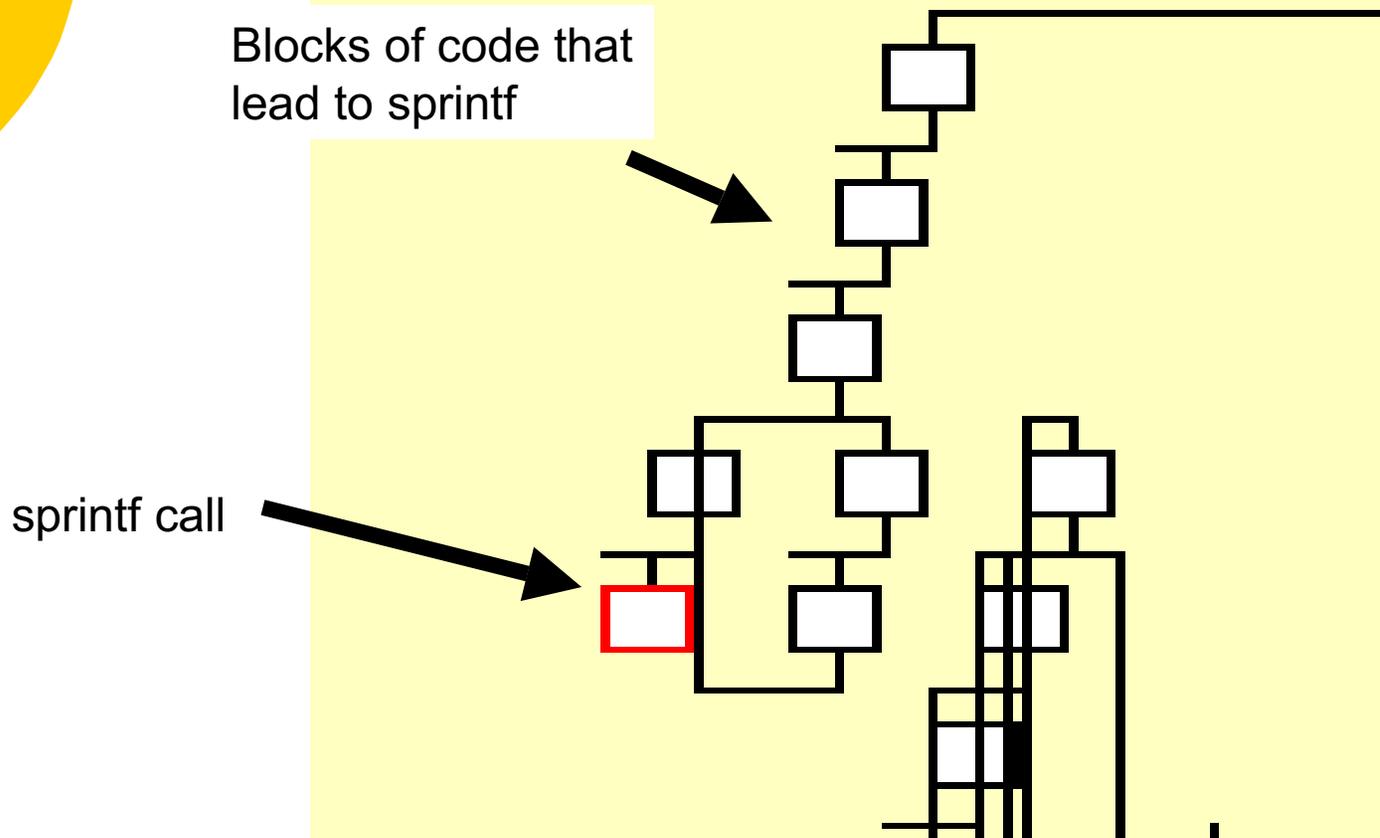


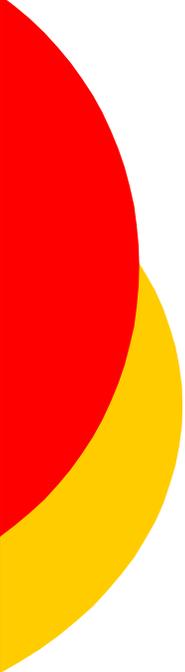
# TEMPEST

---

- Connect the inputs with the bugs
- Verify the exploit
- Build a working exploit
- Offensive stance – find working injection vectors
- Defines a WORKFLOW

# Static backtrace from suspect locations





## 'Self Learning Coverage'

---

- We start with user-supplied data
- We can detect when decisions are calculated from user input
- We can freeze and restore the program at any point and test new values
- Thus, we can map how user-controlled values influence state transitions

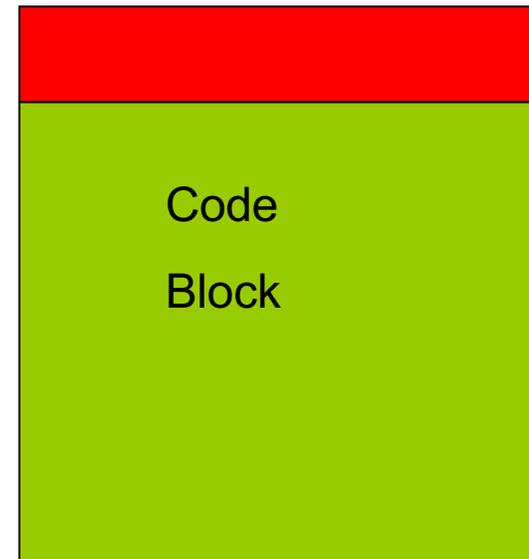


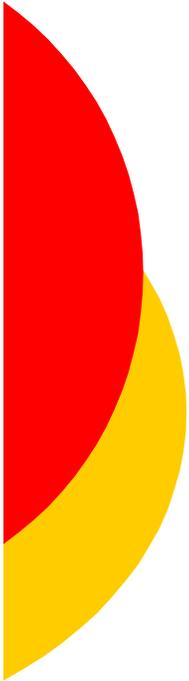
# Location Coverage

---

- As program is used, if a code block is visited it will be highlighted 'grey'\*

Breakpoint

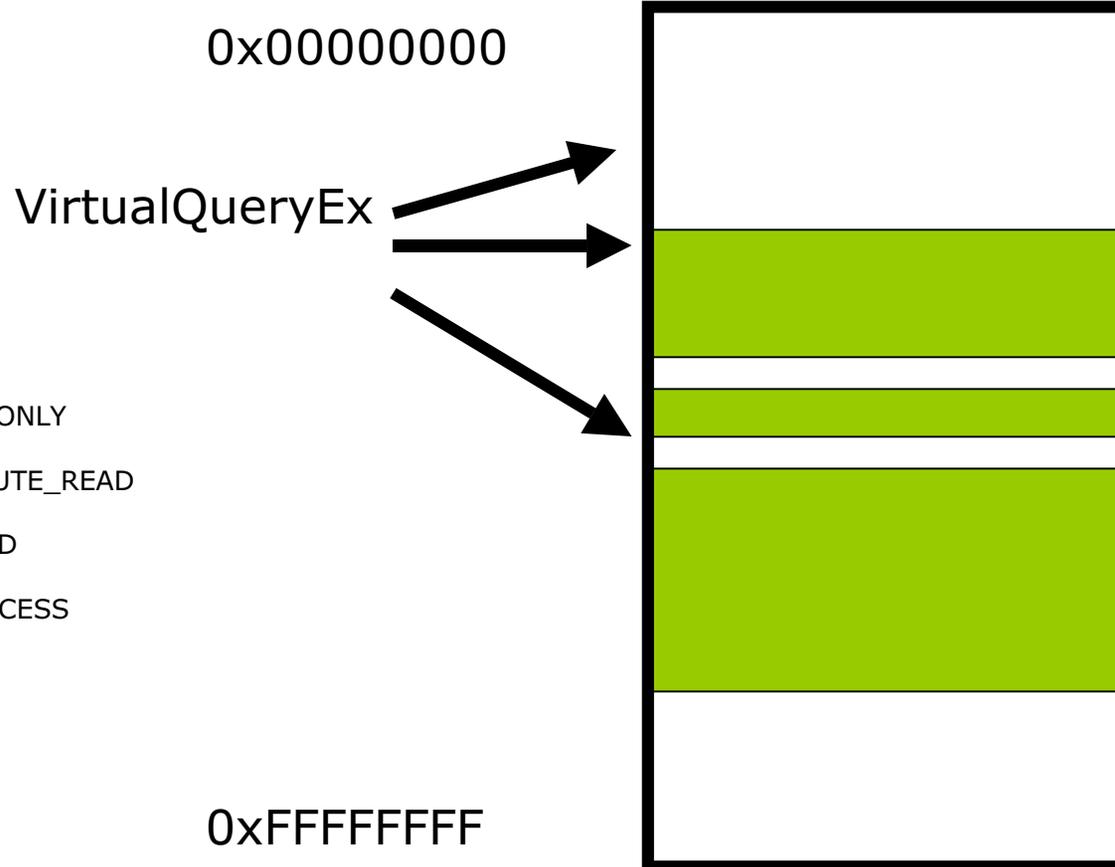




# Process Snapshots

---

MEM\_COMMIT  
&&  
**NOT** PAGE\_READONLY  
&&  
**NOT** PAGE\_EXECUTE\_READ  
&&  
**NOT** PAGE\_GUARD  
&&  
**NOT** PAGE\_NOACCESS



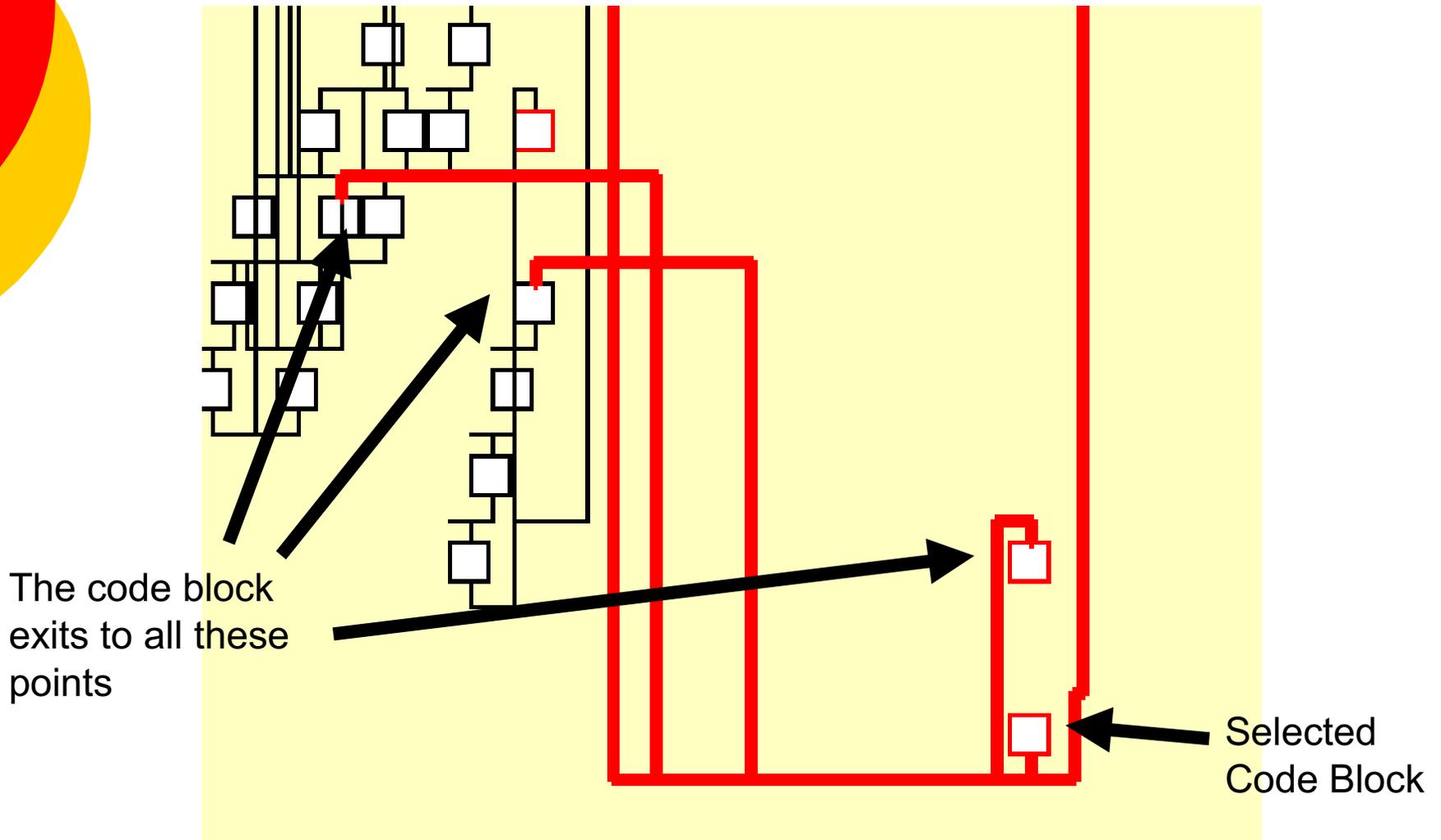


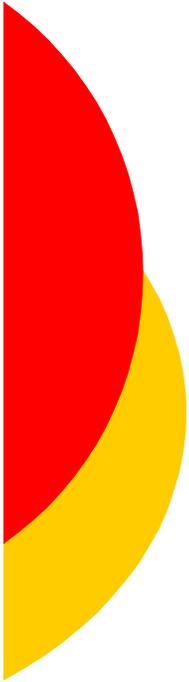
## Fly-By's & Drill Downs

---

- If we hit code blocks 'above' a suspect location we are alerted to potential operations that will cause the target to be exercised
- Coverage helps us tune our input data to drill down to a target location
  - This is the fundamental advantage

# Tracing

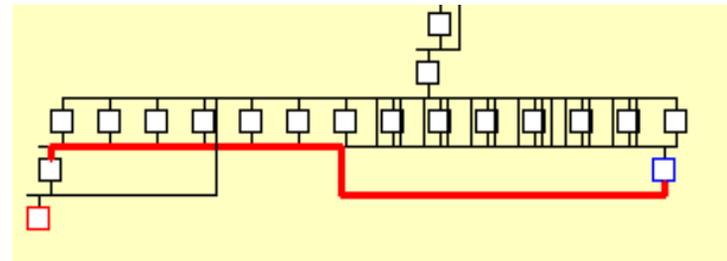
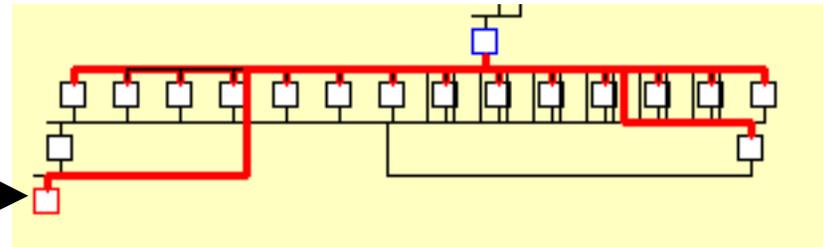
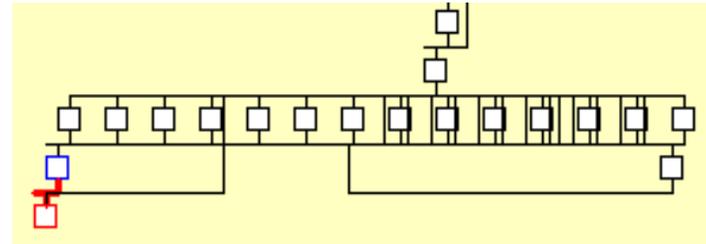
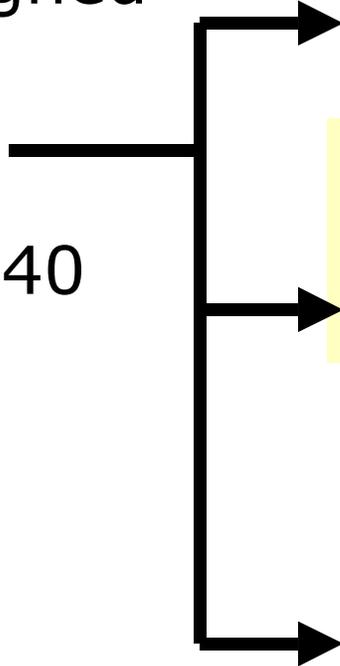


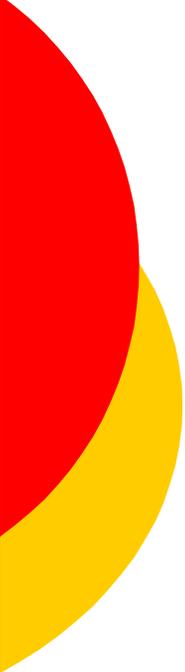


# Trillian IRC DLL

---

Signed/Unsigned mismatch in subroutine at 0x1000FE40





# Boron Tagging

---

- Traces from known points
- Breakpoints on suspect calls
- Can be used as a strategy to skip large sections of the graph
  - These become 'clusters'
  - We cannot create a spanning tree graph unless everything is connected

# Leap Frogging

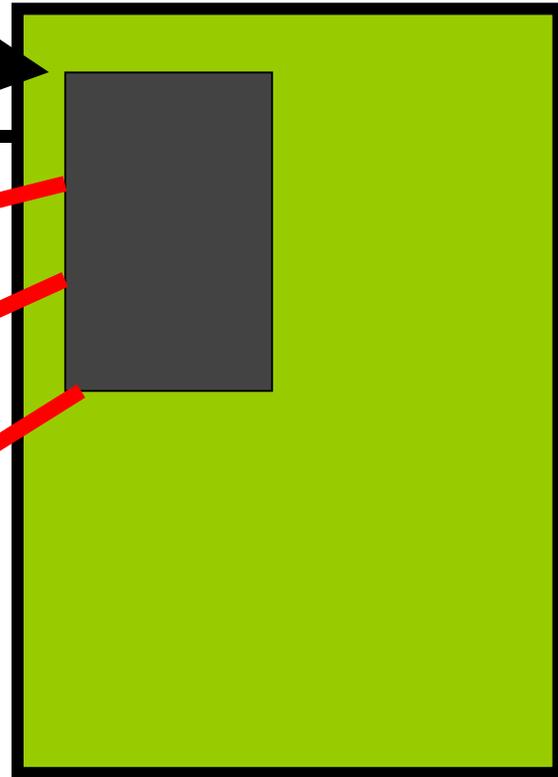
recv( ... )

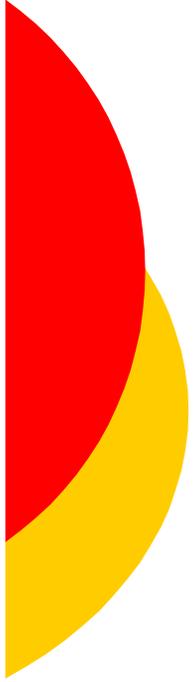
Change page protection  
in order to track access

mov edx, [esi]

mov [ecx], eax

lea edx,  
[esi]



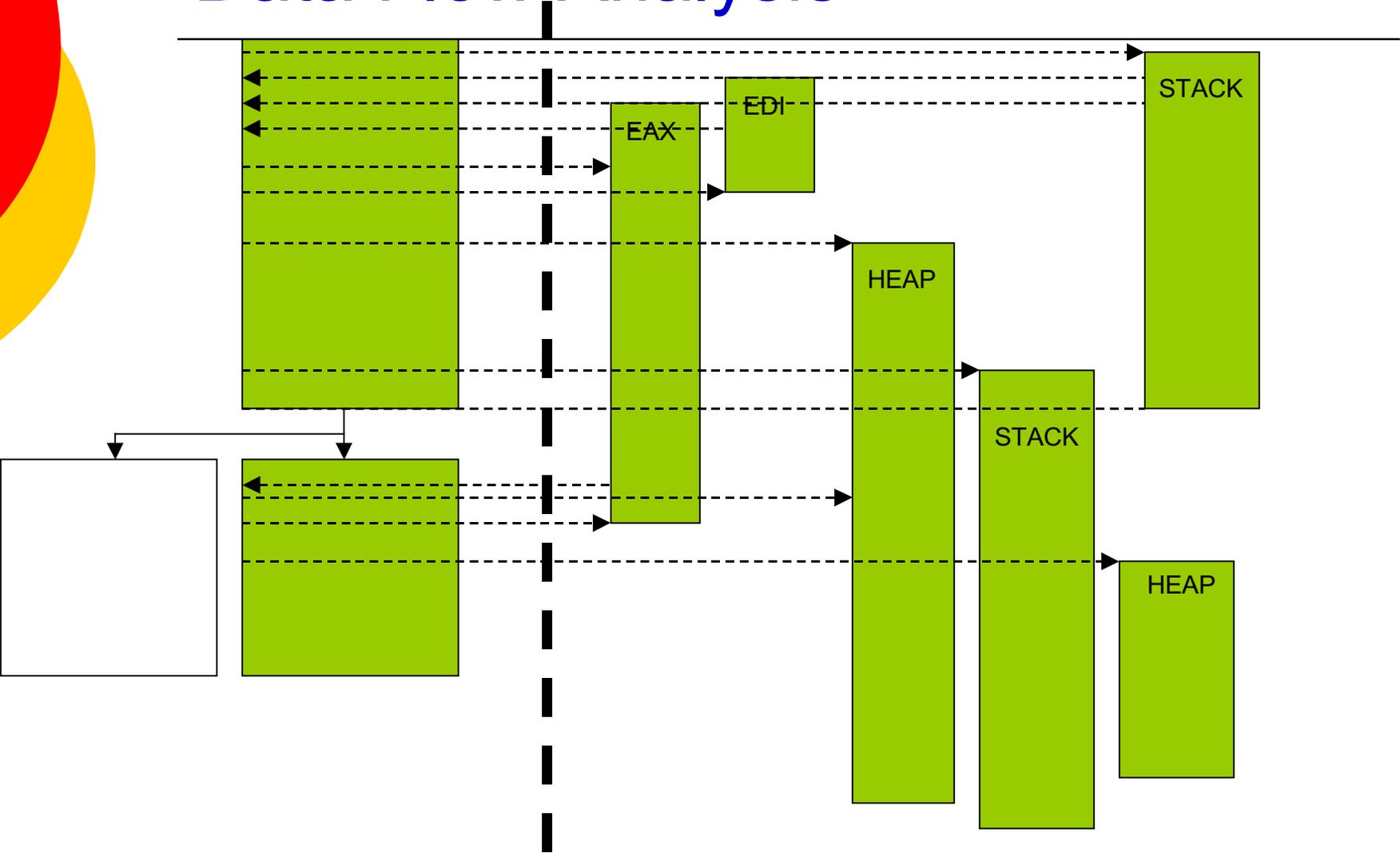
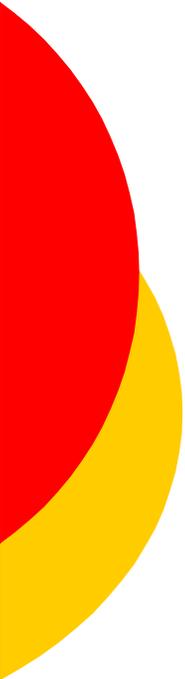


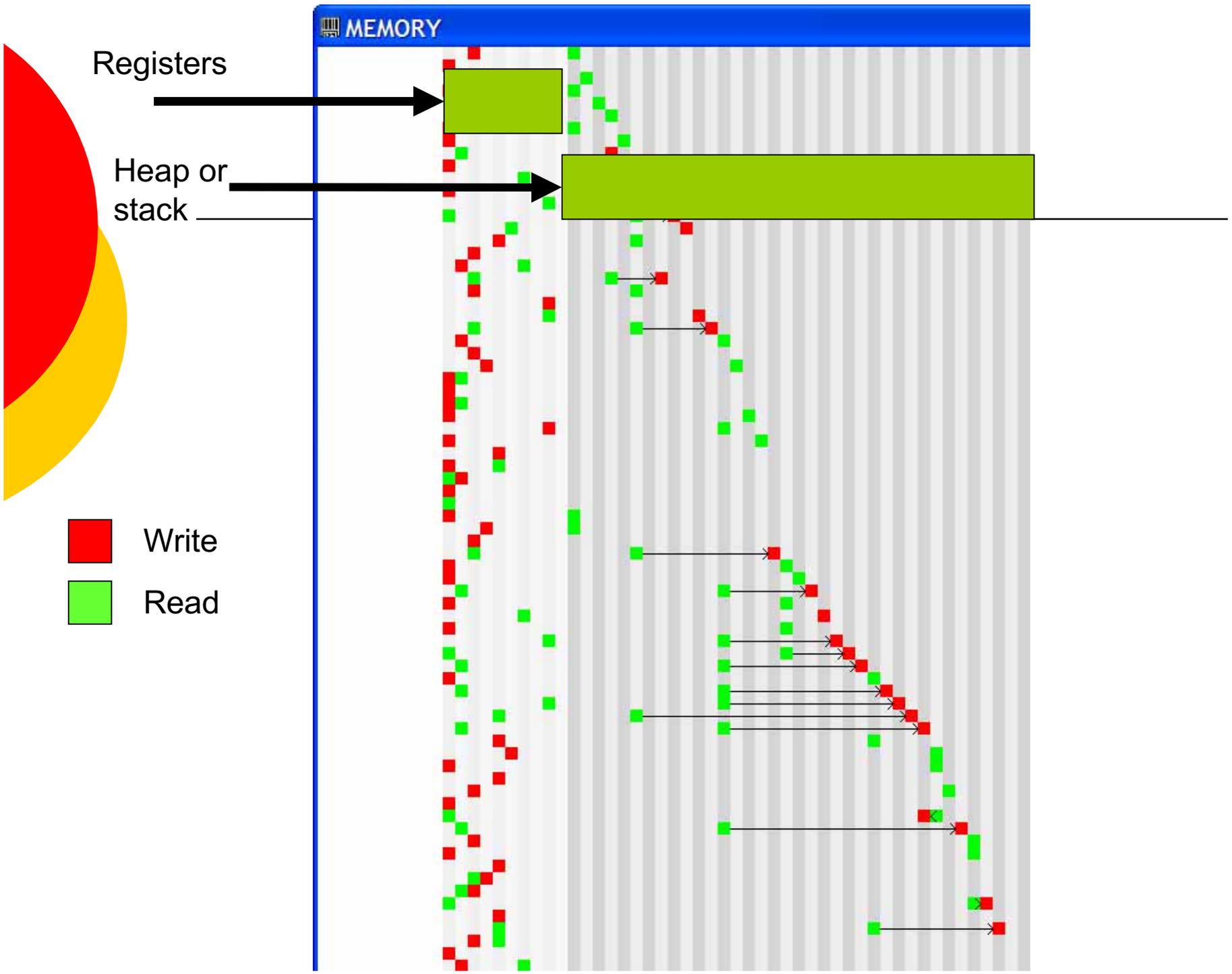
## Leapfrog with Boron

---

- Read memory to find all boron strings
- Set memory breakpoints on all these locations
- Locations are typically re-used
- Doesn't always work because memory is cleared after use

# Data Flow Analysis

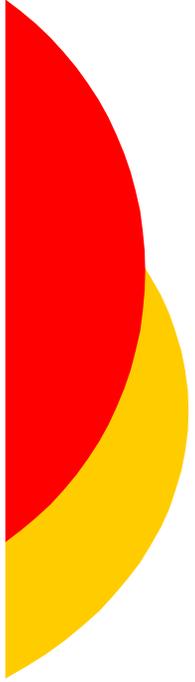




Registers

Heap or stack

Write  
Read



# Graphing Problems

---

- Graph complexity increases with the number of back traces
- Using tempest on more than a few target points at a time results in a huge, unwieldy graph

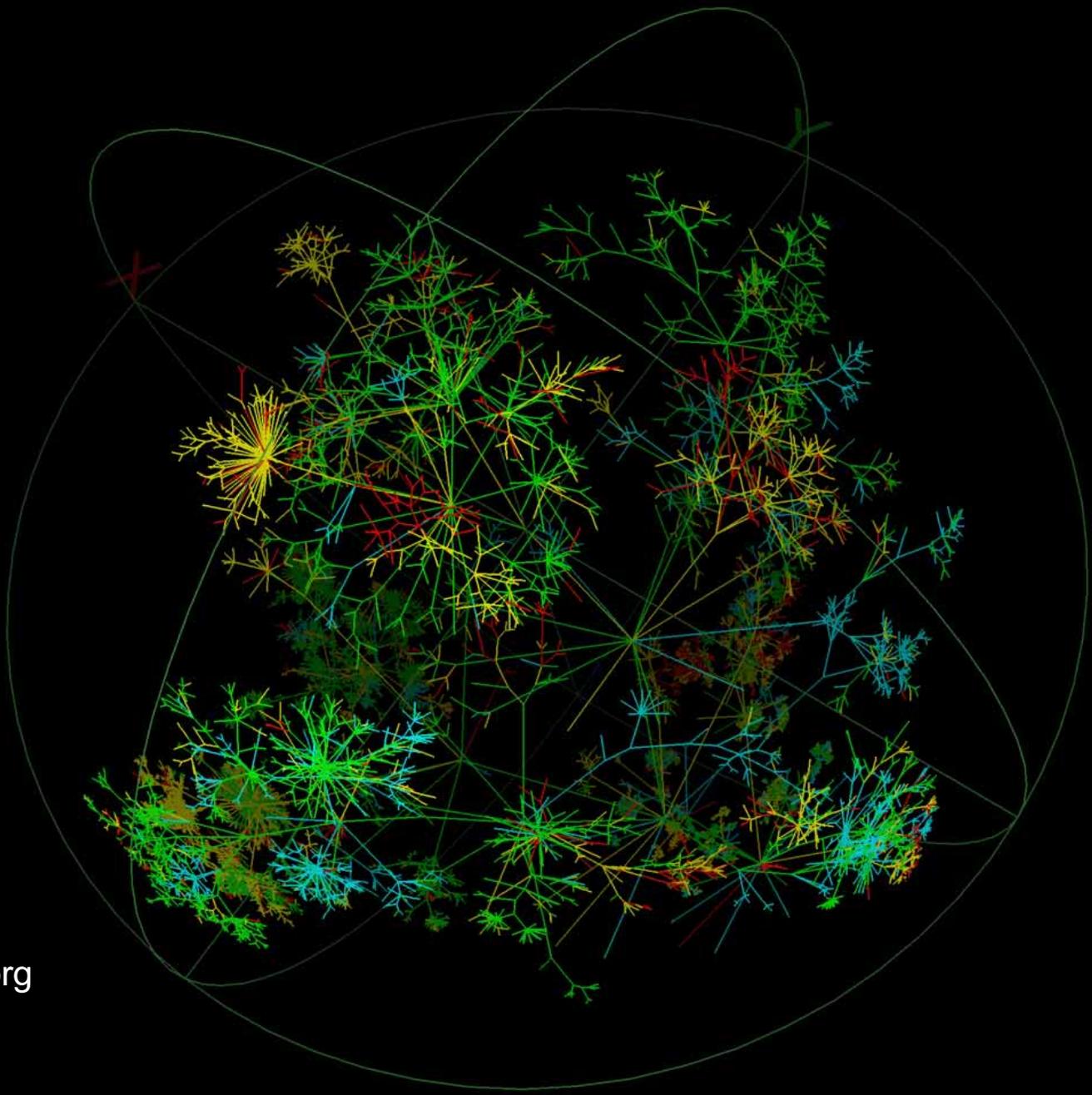


# Advanced Graphing

---

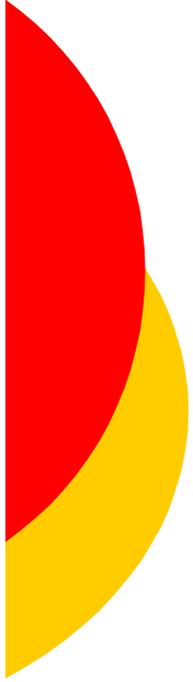
- Different graphing algorithms can be used
- Hyperbolic graphs serve better for browsing a large number of nodes





WALRUS

[www.caida.org](http://www.caida.org)



## Filtering the set

---

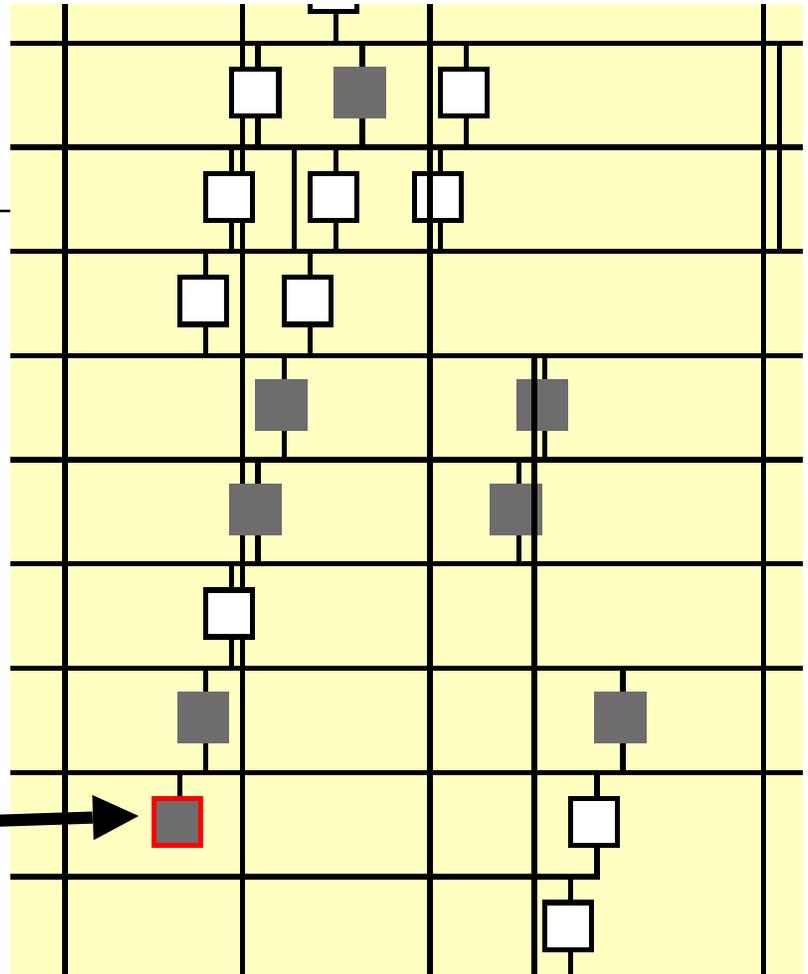
- Don't worry about sprintf if the format string doesn't contain %s
- Don't worry about off by ones if the size parameter is less than the stack correction
- Don't worry about ***anything*** if the source data is not obtained from outside the function





### Work Items

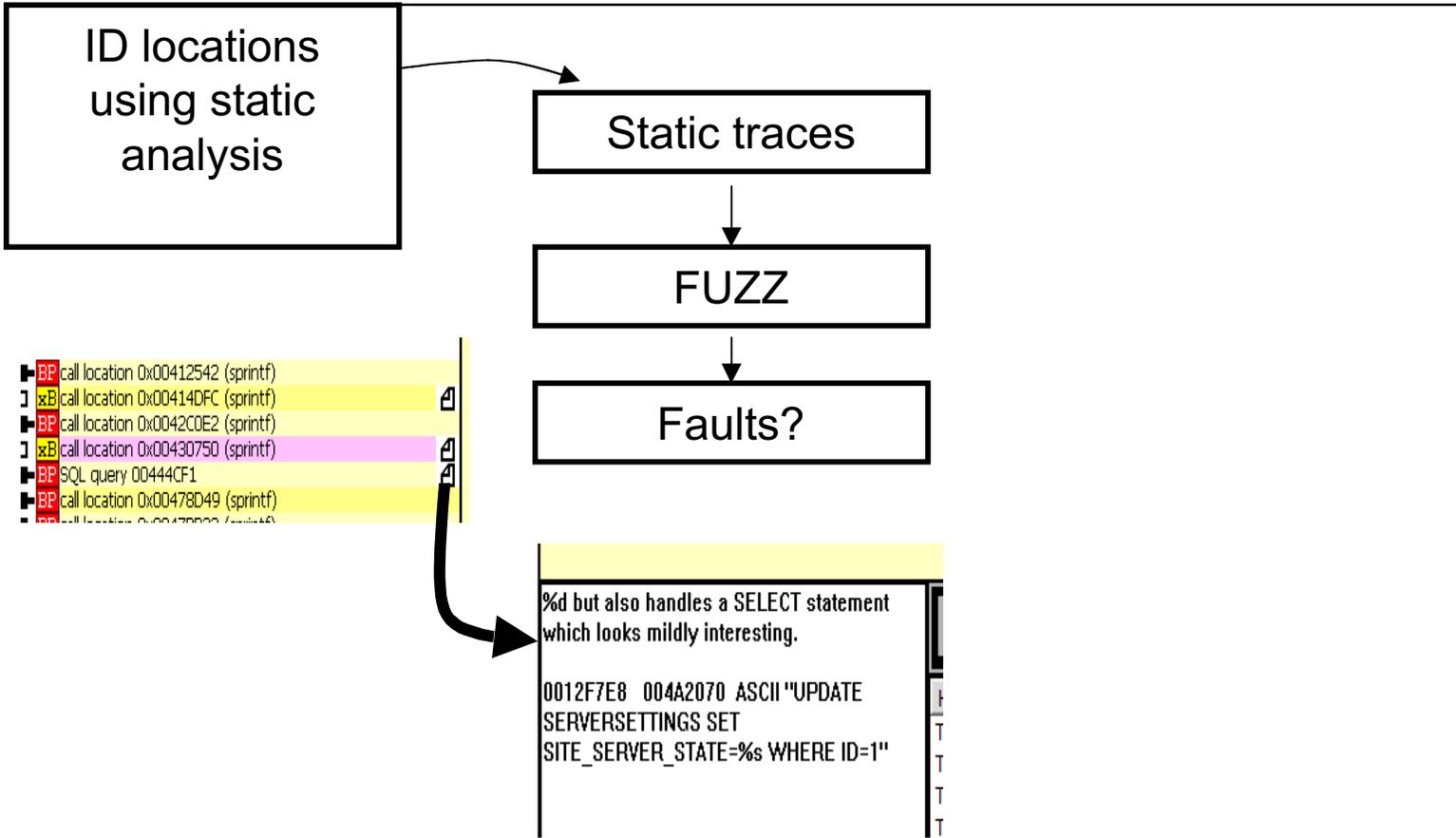
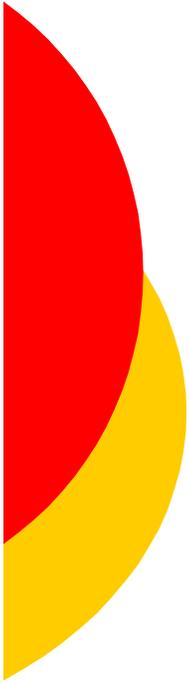
- BP call location 0x00412542 (sprintf)
- xB call location 0x00414DFC (sprintf)
- BP call location 0x0042C0E2 (sprintf)
- xB call location 0x00430750 (sprintf)
- BP SQL query 00444CF1
- BP call location 0x00478D49 (sprintf)
- BP call location 0x0047BB22 (sprintf)
- BP call location 0x0047DA32 (sprintf)
- BP call location 0x00484D27 (sprintf)
- BP call location 0x00484D3B (sprintf)
- BP call location 0x00487103 (sprintf)
- BP call location 0x00487360 (sprintf)



This is a HIT  
- This causes  
a work item to  
be exercised.

Is user-supplied data used in the suspect call?

Hits	
Time: 12:25:57:257	EAX: 08984058(144195672 ) -> SELECT * FROM ACCOUN
Time: 12:25:57:257	EBX: 00B4F0F4(11858164 ) -> .w.  L..
Time: 12:25:57:257	ECX: 00000014(20 )
Time: 12:25:57:257	EDX: 00000014(20 )
Time: 12:25:57:257	ESI: 00B4F7AC(11859884 ) -> X@.  .k>  ...
Time: 12:25:57:257	EDI: 0000002A(42 )
Time: 12:25:57:257	EBP: 004A0604(4851204 ) -> SELECT * FROM GROUPS
Time: 12:25:57:257	ESP: 00B4F0C0(11858112 ) -> X@.       J
Time: 12:25:57:257	+0: 08984058(144195672 ) -> SELECT * FROM ACCOUN
Time: 12:25:57:257	+4: 004A0604(4851204 ) -> SELECT * FROM GROUPS
Time: 12:25:57:257	+8: 00B4F0F4(11858164 ) -> .w.  L..
Time: 12:25:57:257	+12: 77121644(1997674052) -> .D\$  f.
Time: 12:25:57:257	+16: 003E4F50(4083536 ) -> .5J





Faults?



Unresolved Branches?

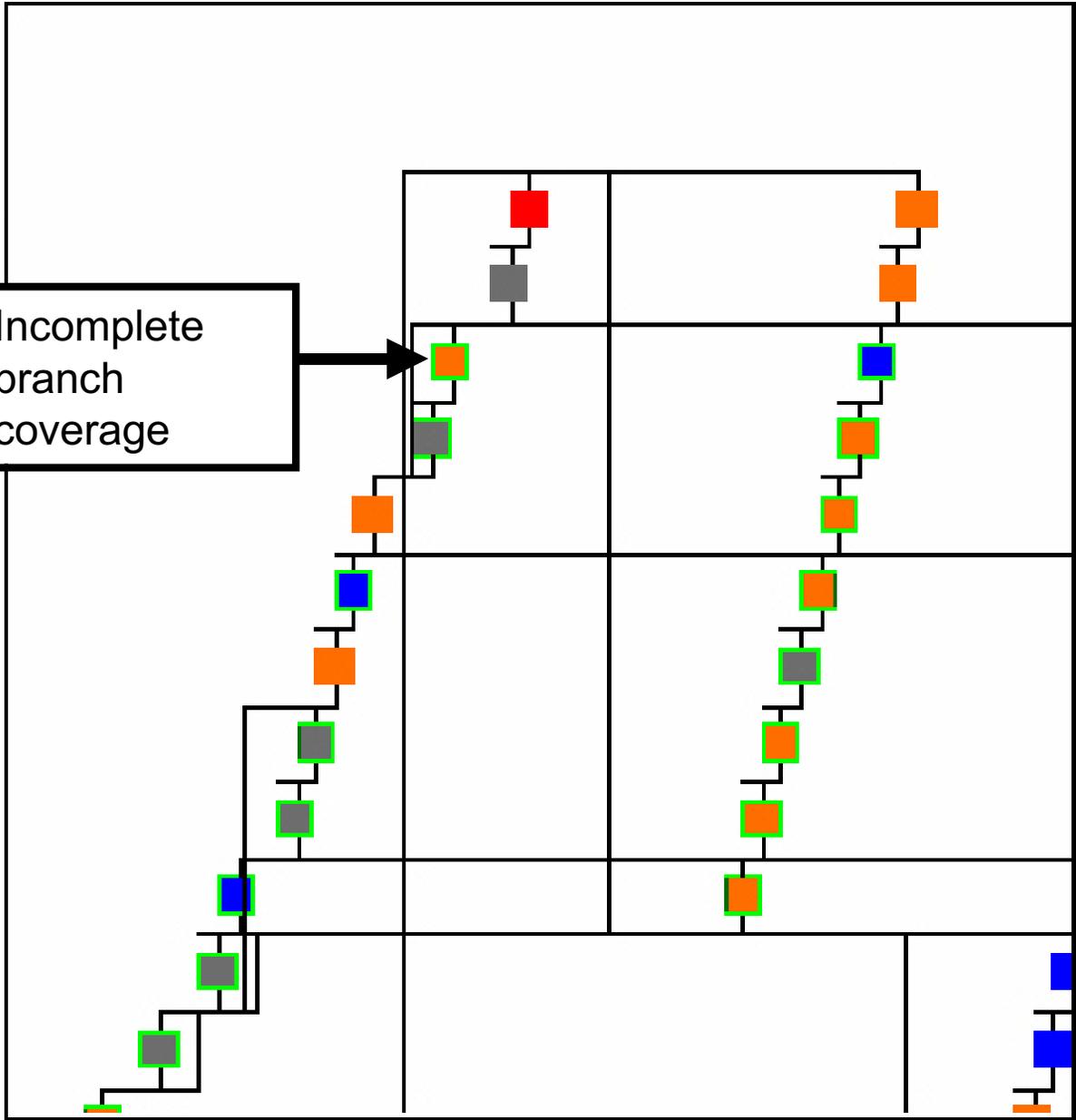


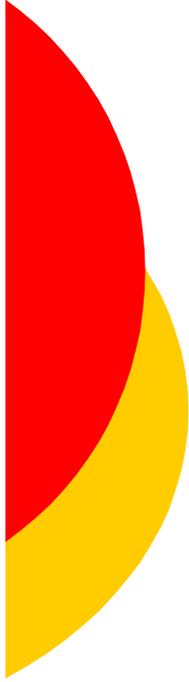
Use Data Flow Analysis to determine if branch is calculated from user-controlled data



Modify input fuzzer to compensate

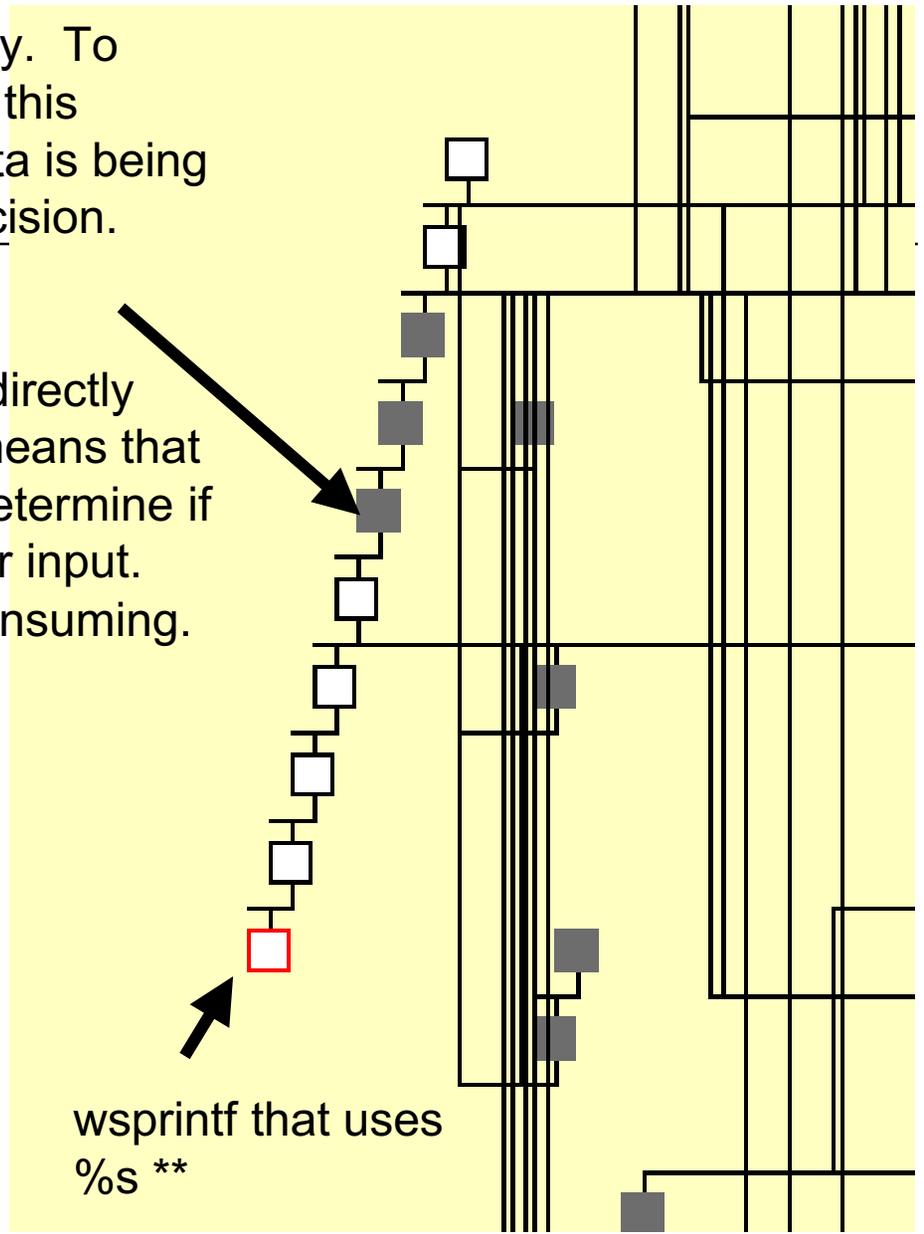
Incomplete branch coverage



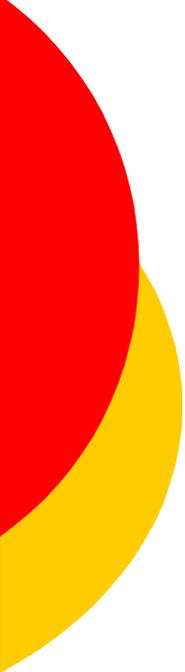


This location is the nearest fly-by. To solve the problem we must visit this location and determine what data is being used to make the branching decision.

In most cases, the value is not directly controlled by the fuzzer. This means that we must trace back further to determine if the value is calculated from user input. This is both tedious and time consuming.



\*\* this graph generated from commercial proxy server (vendor not revealed)



## Conclusion

---

- There exists a process to connect user-input to potential vulnerabilities
- By tracing data and control flow at runtime, a fuzzer can be tuned to target a location
- Only a certain percentage of those bugs identified statically will be exploitable



## Closing Remarks

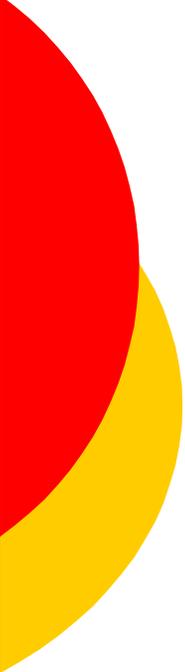
---

BugScan is a commercial product  
that can be obtained from

[www.hbgary.com](http://www.hbgary.com)

Spike is free and can be obtained  
from

[www.immunitysec.com](http://www.immunitysec.com)

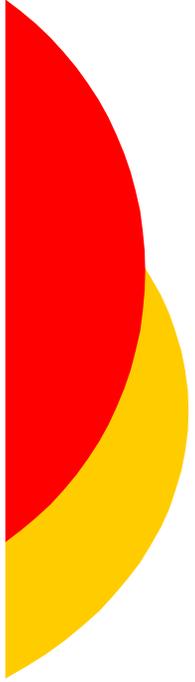


## Closing Remarks

---

- The Tempest debugging system is used internally by HBGary and is not a commercial product
- Many components of the tempest system are open source and can be obtained for study

**[www.hbgary.com](http://www.hbgary.com)**



# Thank You

---

Greg Hoglund  
HBGary, LLC.

[hoglund@hbgary.com](mailto:hoglund@hbgary.com)