DTrace: The Reverse Engineer's Swiss Army Knife

Tiller Beauchamp

David Weston

SAIC

From Science to Solutions

What Is DTrace™?

DTRACE BACKGROUND

*Dtrace was created by Sun Microsystems, Inc. and released under the Common Development and Distribution License (CDDL), a free software license based on the Mozilla Public License (MPL).

DTrace Background

- Kernel-based dynamic tracing framework
- Created by Sun Microsystems
- First released with Solaris™ 10 operating System
- Now included with Apple OS X Leopard
- Soon to be included with FreeBSD
- OpenBSD, NetBSD, Linux?

DTrace Overview

- DTrace is a framework for performance observability and debugging in real time
- Tracing is made possible by thousands of “probes” placed “on the fly” throughout the system
- Probes are points of instrumentation in the kernel
- When a program execution passes one of these points, the probe that enabled it is said to have fired

SAIC
From Science to Solutions

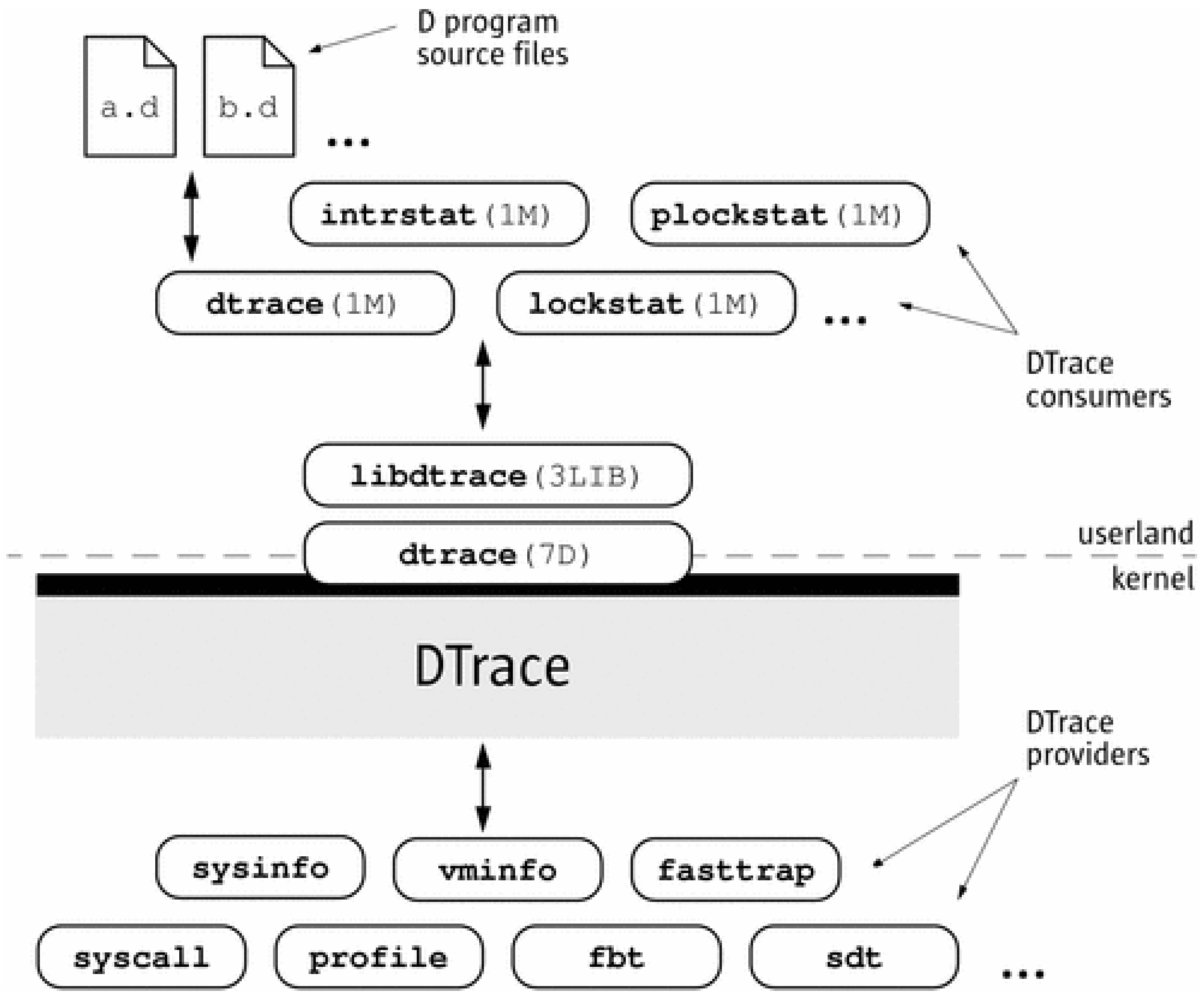
DTrace Overview (cont.)

- DTrace can bind a set of *actions* to each probe
- Actions include stack trace, timestamp, and the argument to a function
- As each probe fires, DTrace gathers the data from your probes and reports it back to you
- If you don’t specify any actions for a probe, DTrace will just take note of each time the probe fires

DTrace Architecture

- DTrace exists almost entirely as a kernel module
- The kernel module performs ALL processing and instrumentation
- DTrace consumers communicate with the kernel module through user module libdtrace
- The `dtrace` command itself is a DTrace consumer because it is built on top of the DTrace library

DTrace Architecture



Source: Solaris Dynamic Tracing Guide

The D Language

- D is an interpreted, block-structured language
- D syntax is a subset of C
- D programs are compiled into intermediate form
- Intermediate form is validated for safety when your program is first examined by the DTrace kernel software
- The DTrace execution environment handles any runtime errors

The D Language

- D does not use control-flow constructs such as if statements and loops
- D program clauses are written as single, straight-line statement lists that trace an optional, fixed amount of data
- D can conditionally trace data and modify control flow using logical expressions called *predicates*
- *A predicate is tested at probe firing before executing any statements*

SAIC
From Science to Solutions

DTrace Features

- DTrace is dynamic: probes are enabled only when you need them
- No code is present for inactive probes
- There is no performance degradation when you are not using DTrace
- When the `dtrace` command exits, all probes are disabled and instrumentation removed
- The system is returned to its original state

DTrace Features (cont.)

- DTrace is nondestructive. The system is not paused or quiesced
- DTrace is designed to be efficient. No extra data are ever traced
- Because of its safety and efficiency, DTrace can be used in production to solve real problems in real time
- These same features can be leveraged by reverse engineers and security researchers

DTrace Uses

- DTrace takes the power of multiple tools and unifies them with one programmatically accessible interface
- DTrace has features similar to the following:
  - `truss`: tracing system calls, user functions
  - `ptrace`: tracing library calls
  - `prex/tnf*`: tracing kernel functions
  - `lockstat`: profiling the kernel
  - gdb: access to kernel/user VM

DTrace Uses

- DTrace combines system performance statistics, debugging information, and execution analysis into one tight package
- A real “Swiss army knife” for reverse engineers
- DTrace probes can monitor every part of the system, giving “the big picture” or zooming in for a closer look
- Can debug “transient” processes that other debuggers cannot

Creating DTrace Scripts

- Dozens of ready-to-use scripts are included with Sun's DTraceToolkit; they can be used as templates
- These scripts provide functions such as syscalls by process, reads and writes by process, file access, stack size, CPU time, memory r/w and statistics
- Complex problems can often be diagnosed by a single "one-liner" DTrace script

Example: Syscall Count

- System calls count by application:
  - dtrace -n 'syscall:::entry{@[execname] = count();}'.

```
Matched 427 probes
Syslogd                    1
DirectoryService           2
Finder                     3
TextMate                   3
Cupsd                      4
Ruby                    4309
vmware-vmx              6899
```

Example: File Open Snoop

```
#!/usr/sbin/dtrace -s

syscall::open*:entry {
  printf("%s %s\n",
          execname,
          copyinstr(arg0));
}
```

Example: File Snoop Output

```
vmware-vmx        /dev/urandom
Finder            /Library/Preferences/SystemConfiguration/com.apple.smb.server.plist
iChat             /Library/Preferences/SystemConfiguration/com.apple.smb.server.plist
Microsoft Power   /Library/Preferences/SystemConfiguration/com.apple.smb.server.plist
nmblookup         /System/Library/PrivateFrameworks/ByteRange ... ByteRangeLocking
nmblookup         /dev/dtracehelper
nmblookup         /dev/urandom
nmblookup         /dev/autofs_nowait
Nmblookup         /System/Library/PrivateFrameworks/ByteRange... ByteRangeLocking
```

DTrace Lingo

- Providers pass the control to DTrace when a probe is enabled
- Examples of providers include syscall, lockstat, fbt, io, mib
- Predicates are D expressions
- Predicates allow actions to be taken only when certain conditions are met
- Actions are taken when a probe fires
- Actions are used to record data to a DTrace buffer

DTrace Syntax

- DTrace probe syntax:
  - provider:module:function:name
    - **Provider.** The name of the DTrace provider that created this probe
    - **Module.** The name of the module to which this probe belongs
    - **Function.** The name of the function with which this probe is associated
    - **Name.** The name component of the probe. It generally gives an idea of its meaning

DTrace Syntax

Generic D Script

```
Probe:      provider:module:function:name
Predicate:    /some condition that needs to happen/
                  {
                          actions to act upon
                  }
```

Even sophisticated D scripts can be implemented with just a few probes

How Can We Use DTrace?

DTRACE AND REVERSE ENGINEERING (RE)

DTrace for RE

- DTrace is extremely versatile and has many applications for RE
- It is very useful for understanding the way a process works and interacts with the rest of the system
- DTrace probes work in a manner very similar to debugger “hooks”
- DTrace probes can be much more useful than debugger hooks because they can be described generically and focused later

DTrace for RE

- One of Dtrace’s greatest assets is speed
- DTrace can instrument any process on the system without starting or stopping it (i.e., debuggers)
- Complex operations can be understood with short succinct scripts (i.e., DTrace one-liners)
- You can refine your script as the process continues to run
- Think of DTrace as a rapid development environment for RE tasks

SAIC
From Science to Solutions

DTrace vs. Debuggers

- User mode and kernel mode debuggers allow you to control execution and inspect process information
- DTrace can instrument BOTH MODES AT THE SAME TIME
- To trace execution, debuggers use INT3 instructions to pause and resume execution
- For security researchers and exploit developers, this can cause “phantom exceptions” that can be difficult to troubleshoot

DTrace vs. Debuggers

- DTrace is not a good choice for destructive actions
- DTrace cannot directly alter execution path or change memory values (it can set a breakpoint and transfer control to a debugger)
- DTrace does not directly perform exception handling (you must implement your own)
- Currently DTrace is not susceptible to traditional anti-debugging techniques (isdebuggerpresent()), though Apple has implement probe blocking (just recompile libdtrace with your own patch)

DTrace vs. Tracers

- Truss operates one process at a time, with no systemwide capability
- Truss reduces application performance
- Truss stops threads through procfs, records the arguments for the system call, and then restarts the thread
- Valgrind™ is limited to the user space and cannot gather system statistics
- Ptrace() is much more efficient at instruction level tracing but it is crippled on OS X

*Valgrind is Open Source/Free Software and is freely available under the GNU General Public License.

DTrace Limitations

- The D language does not have conditionals or loops
- The output of many functions is to stdout (i.e., stack(), unstack())
- DTrace has very limited capacity for storing or manipulating data
- We can fix this

SAIC
From Science to Solutions

Ruby DTrace

- To make DTrace even more powerful, we can combine it with an object oriented programming (OOP) scripting language (Ruby, Python)
- Chris Andrews wrapped libdtrace in Ruby, allowing us to create hybrid D/Ruby scripts
- This opens up new possibilities for controlling the process and manipulating the data it returns
- Ruby-DTrace is in the same category as programmable debuggers (pyDBG, knoxDBG, immDBG)

The Power of Ruby

- Ruby-DTrace can be the “glue” in a powerful reverse engineering framework
- By leveraging existing Ruby packages, we have everything we need to do most RE tasks
- idaRub/rublib allows us to harness the power of the powerful IDA disassembler in our Ruby-DTrace script (over the network!@#@#)
- We can use Ruby graphing libraries to perform data visualization (graphviz, openGL)

Ruby-DTrace and Code Coverage

- DTrace can “hook” every function in a process
- This makes it perfect for incrementing a fuzzer with code coverage
- Code coverage is useful for understanding what areas are being fuzzed
- Current RE code coverage monitors are only block based (PaiMei)
- With Ruby-DTrace we can use IDA to obtain block information or check code coverage at the function or instruction level

Ruby-DTrace and Code Coverage

- With IdaRub we can “visualize” code coverage by coloring IDA graph nodes and code paths
- This is extremely powerful when used in conjunction with static analysis
- DTrace allows us to stop and start tracing at any point, all without restarting the target
- Using Ruby packages, we can create many different types of graphs, from block graphs to 3-D (graphviz, OpenGL)

Ruby-DTrace and Exploit Dev

- Ruby-DTrace is adept at understanding vulnerabilities and exploiting them
- DTrace probes allow you to track data input flow throughout a process to understand where and why memory corruption took place
- Vulnerability analysis times of conventional debuggers can be dramatically reduced with Ruby-DTrace and IdaRub
- Methods that cause stack and heap corruption can be pinpointed with IDA comments or coloring

Helpful Features

DTrace gives us some valuable features for free:

- Control flow indicators
- Symbol resolution
- Call stack trace
- Function parameter values
- CPU register values

Control Flow

```
1        -> -[AIContentController finishSendContentObject:]
1          -> -[AIAdium notificationCenter]
1          <- -[AIAdium notificationCenter]
1          -> -[AIContentController processAndSendContentObject:]
1            -> -[AIContentController handleFileSendsForContentMessage:]
1            <- -[AIContentController handleFileSendsForContentMessage:]
1            -> -[AdiumOTREncryption willSendContentMessage:]
1              -> policy_cb
1                -> contactFromInfo
1                  -> -[AIAdium contactController]
1                  <- -[AIAdium contactController]
1                  -> accountFromAccountID
```

Symbol and Stack Trace

```
dyld`strcmp
  dyld`ImageLoaderMachO::findExportedSymbol(char
  dyld`ImageLoaderMachO::resolveUndefined(...
  dyld`ImageLoaderMachO::doBindLazySymbol(unsigned
  dyld`dyld::bindLazySymbol(mach_header const*, ...
  dyld`stub_binding_helper_interface2+0x15
  Ftpd`yylex+0x48
  Ftpd`yyparse+0x1d5
  ftpd`ftp_loop+0x7c
  ftpd`main+0xe46
```

Function Parameters

DTrace's copyin* functions allow you to copy data from the process space:

```
printf("arg0=%s", copyinstr( arg0 ))
```

Output:

```
 1  -> strcmp     arg0=_isspecial_l
```

CPU Register Values

Uregs array allows access to reading CPU registers

```
  printf("EIP:%x", uregs[R_EIP]);
```

Example:

```
  EIP: 0xdeadbeef
  EAX: 0xffffeae6
  EBP: 0xdefacedd
  ESP: 0x183f6000
```

Destructive Examples

```
#!/usr/sbin/dtrace -w -s
syscall::uname:entry { self->a = arg0; }

syscall::uname:return{
        copyoutstr(“Windows”, self->a, 257);
        copyoutstr(“PowerPC”, self->a+257, 257);
        copyoutstr(“2010.b17”, self->a+(257*2), 257);
        copyoutstr(“fud:2010-10-31”, self->a+(257*3), 257);
        copyoutstr(“PPC”, self->addr+(257*4), 257);
}
```

Adapted from: Jon Haslam, http://blogs.sun.com/jonh/date/20050321

Snooping

```
syscall::write: entry {
   self->a = arg0;
}
syscall::write: return {
   printf(“write: %s”,
   copyinstr(self->a);
}
```

Got Ideas?

Using DTrace:

- Monitor stack overflows
- Code coverage
- Fuzzer feedback
- Monitor heap overflows

Writing a Stack Overflow Monitor

MONITORING THE STACK

Stack Overflow Monitoring

Programmatic control at EIP overflow time allows you to:

- Pinpoint the vulnerable function
- Reconstruct the function call trace
- Halt the process before damage occurs (HIDS)
- Dump and search process memory
- Send feedback to fuzzer
- Attach debugger
- Attempt repair (?)

Overflow Detection in One Probe

```
#/usr/sbin/dtrace -w -s

pid$target:::return
    / uregs[R_EIP] == 0x41414141 / {
    printf("Don’t tase me bro!!!");
             stop()
             ...
}
```

Cautionaries

A few issues to be aware of:

- DTrace drops probes by design
- Tune options, narrow trace scope to improve performance
- Some libraries and functions behave badly
- Stack overflows can cause violations before function return

First Approach

- Store RETURN value at function entry
- uregs[R_SP], NOT uregs[R_ESP]
- Compare EIP to saved RETURN value at function return
- If different, there was an overflow

Simple enough, but false positives from:

- Tail call optimizations
- Functions without return probes

DTrace and Tail Calls

C calls A, which returns value of call to B

```
func A(int x, int y){
  .....
  return B(x, y);
}
```

- Compiler optimizes by letting B use A’s frame
- Saves resources and requires fewer instructions

DTrace and Tail Calls (cont.)

- DTrace reports tail calls as a return from A and an entry to B
- This is as if function C called A and then called B
- EIP on return from A is first instruction of B, NOT the next instruction in C
- Saved RETURN ! = EIP on return of A

New Approach

- Store RETURN value at function entry
- At function return, compare saved RETURN value with CURRENT value
- Requires saving both the original return value and its address in memory
- Fires when saved RETURN ! = current RETURN and EIP = current RETURN

But Missing Return Probes???

Still trouble with functions that “never return”

- Some functions misbehave
- DTrace does not like function jump tables (dyld_stub_*)
- Entry probe but no exit probe

Determining Missing Returns

Using DTrace – l flag

- List entry/exit probes for all functions
- Find functions with entry but no exit probe

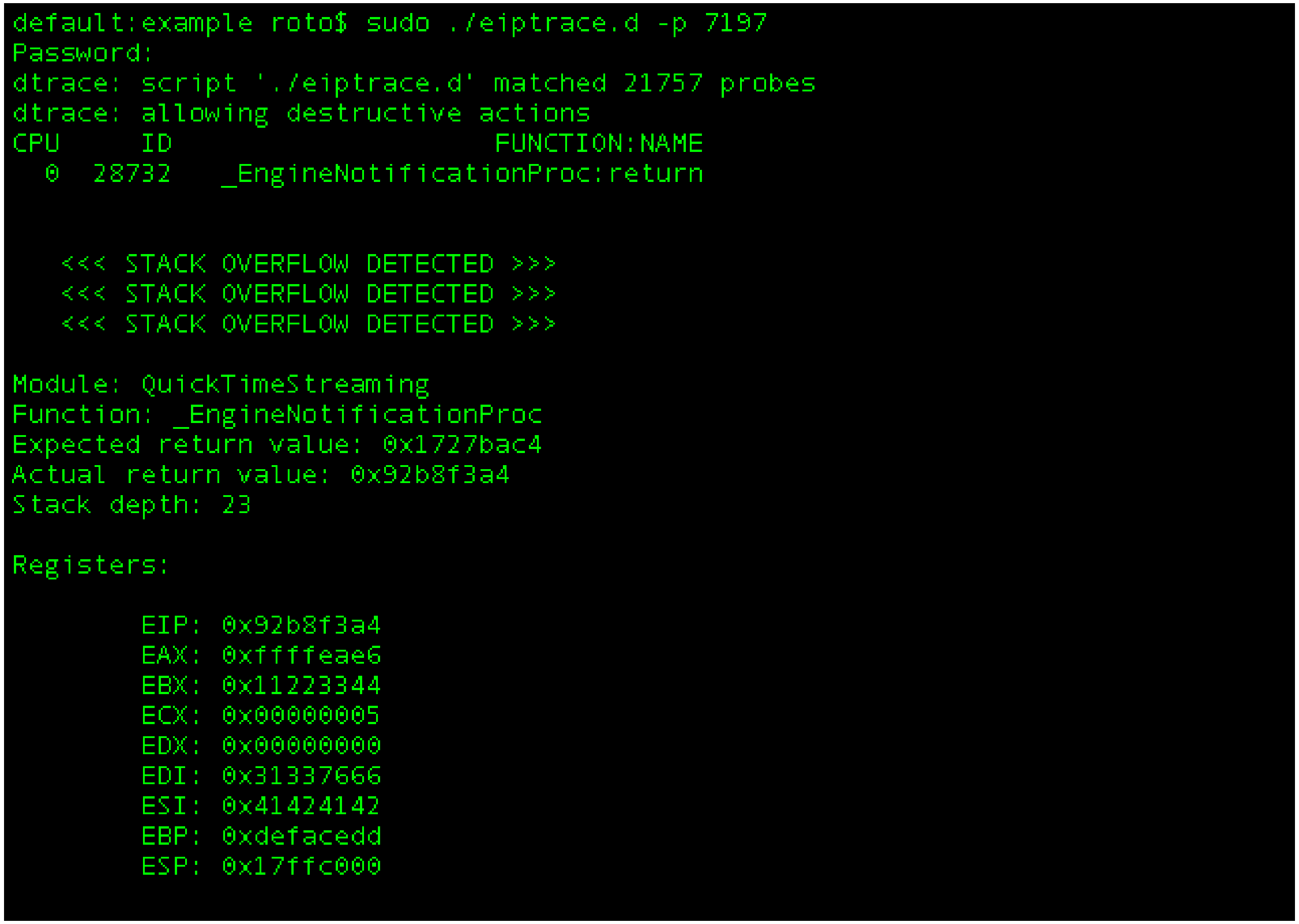
Using DTrace aggregates

- Run application
- Aggregate on function entries and exits
- Look for mismatches

Exclude these functions with predicates

- / probefunc ! = “everybodyJump” /

Stack Overflow in Action



```
default:example roto$ sudo ./eiptrace.d -p 7197
Password:
dtrace: script './eiptrace.d' matched 21757 probes
dtrace: allowing destructive actions
CPU     ID                    FUNCTION:NAME
  0  28732   _EngineNotificationProc:return


   <<< STACK OVERFLOW DETECTED >>>
   <<< STACK OVERFLOW DETECTED >>>
   <<< STACK OVERFLOW DETECTED >>>

Module: QuickTimeStreaming
Function: _EngineNotificationProc
Expected return value: 0x1727bac4
Actual return value: 0x92b8f3a4
Stack depth: 23

Registers:

        EIP: 0x92b8f3a4
        EAX: 0xffffeae6
        EBX: 0x11223344
        ECX: 0x00000005
        EDX: 0x00000000
        EDI: 0x31337666
        ESI: 0x41424142
        EBP: 0xdefacedd
        ESP: 0x17ffc000
```

Advanced Tracing

Diving in deeper:

- Instruction-level tracing
- Code coverage with IDA Pro and IdaRub
- Profiling idle and GUI code
- Feedback to the fuzzer, smart/evolutionary fuzzing
- Conditional tracing based on function parameters (reaching vulnerable code paths)

Instruction Tracing

CODE COVERAGE

Code Coverage Approach

Approach

- Instruction-level tracing using DTrace
- No breakpoints required
- Minimal (really?) interference with application
- Use IdaRub to send commands to IDA
- IDA colors instructions and code blocks
- Can be done in real time, if you can keep up

Tracing Instructions

- The last field of a probe is the offset in the function
- Entry = offset 0
- Leave blank for every instruction
- Must map static global addresses to function offset addresses

Print address of every instruction:

```
pid$target:a.out:: { print(“%d”, uregs[R_EIP]); }
```

Ruby-DTrace

- Wraps libdtrace
- Not much different than parsing output of DTrace
- Communication is one way
- Buys us the programmatic response
- Still does not interfere with application/kernel

IdaRub

- Wraps IDA interface
- Ruby code is the client
- Server is IDA plugin
- Ruby glues it all together
- IdaRub was released by Spoonm at REcon 2006

```
ida.set_item_color(eip, 3000)
```

More info:
http://www.metasploit.com/users/spoonm/idarub/

Library Coverage

- Must know memory layout of program
- `vmmap` on OS X will tell you
- Use offset to map runtime library EIPs to decompiled libraries

Code Coverage with DTrace

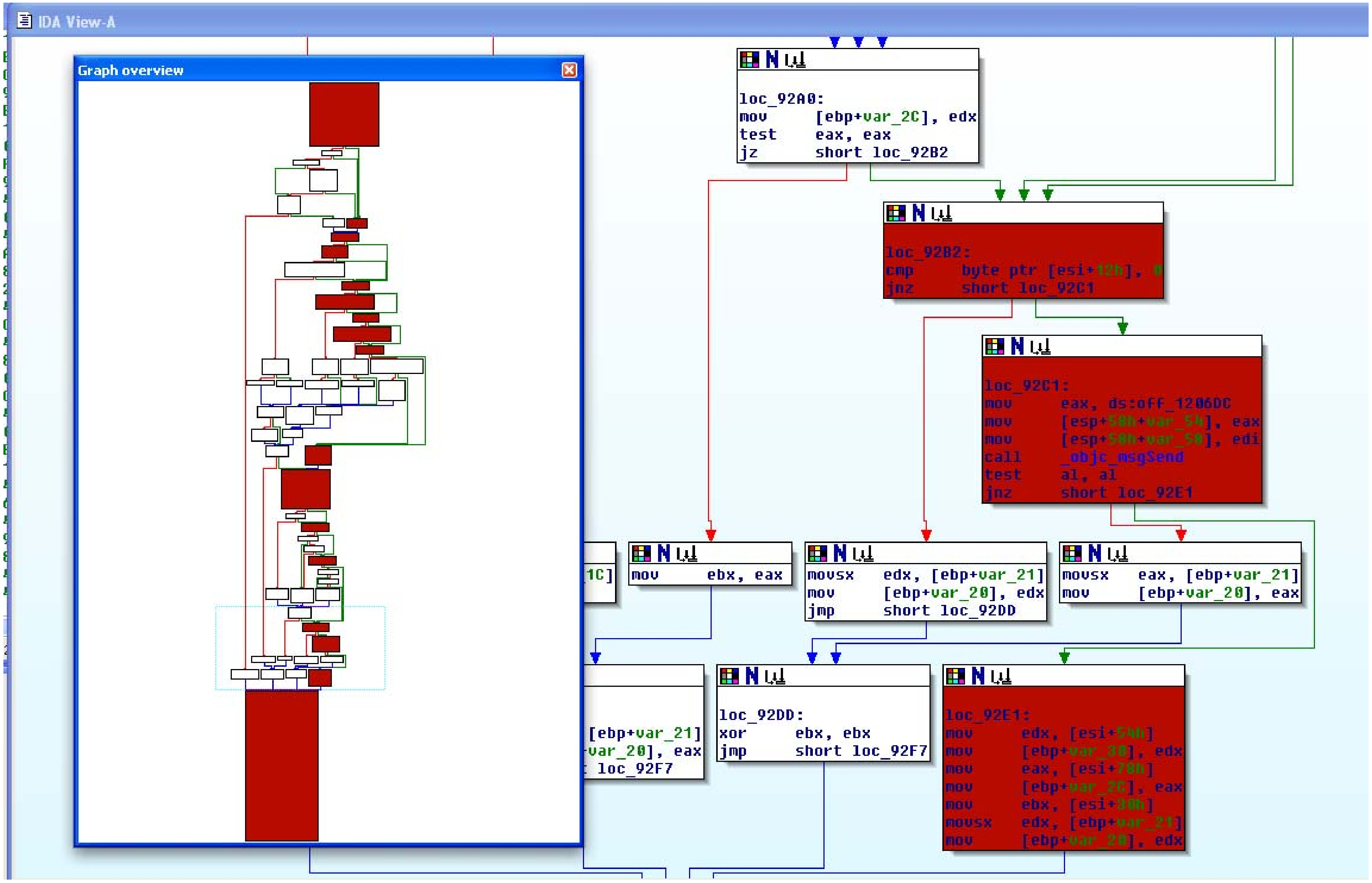
Capabilities:

- Associate fuzz runs with code hit
- Visualize code paths
- Record number of times blocks were hit
- Compare idle traces to other traces

Limitations:

- Instruction tracing can be slow for some applications
- Again, tuning and limiting scope

Coverage Visualization



```
loc_92A0:
mov     [ebp+var_2C], edx
test    eax, eax
jz      short loc_92B2
```

```
loc_92B2:
cmp     byte ptr [esi+12h], 0
jnz     short loc_92C1
```

```
loc_92C1:
mov     eax, ds:off_1206DC
mov     [esp+58h+var_54], eax
mov     [esp+58h+var_58], edi
call    _objc_msgSend
test    al, al
jnz     short loc_92E1
```

```
1C]
```

```
mov     ebx, eax
```

```
movsx   edx, [ebp+var_21]
mov     [ebp+var_20], edx
jmp     short loc_92DD
```

```
movsx   eax, [ebp+var_21]
mov     [ebp+var_20], eax
```

```
[ebp+var_21]
var_20], eax
loc_92F7
```

```
loc_92DD:
xor     ebx, ebx
jmp     short loc_92F7
```

```
loc_92E1:
mov     edx, [esi+54h]
mov     [ebp+var_30], edx
mov     eax, [esi+78h]
mov     [ebp+var_2C], eax
mov     ebx, [esi+30h]
movsx   edx, [ebp+var_21]
mov     [ebp+var_20], edx
```

Writing a Heap Overflow Monitor

MONITORING THE HEAP

Ruby-DTrace and the Heap

- The heap has become “the” major attack vector replacing stack-based buffer overflows
- Relatively common unlink() write4 primatives are no longer as “easy” to exploit
- See Aitel and Waisman’s excellent “Debugging with ID” presentation for more details
- As they point out, the key to the “new breed” of heap exploit is understanding the heap layout and allocation patterns
- ImmDBG can help you with this on XP, and Gerrado Richarte’s heap tracer can help you with this on Solaris and Linux

Ruby-DTrace and the Heap

- Just as heap checks have killed unlink4() on >= SP2 and glibc
- Nemo’s (Phrack 64) technique for overwriting the malloc_zone_t function pointers died with Tiger 10.4.2
- Exploit engineers need help on OS X
- RE:Trace can help you on OS X
- DTrace naturally hooks the functions necessary to understand the layout of the heap and its allocation patterns
- With DTrace we can do even more

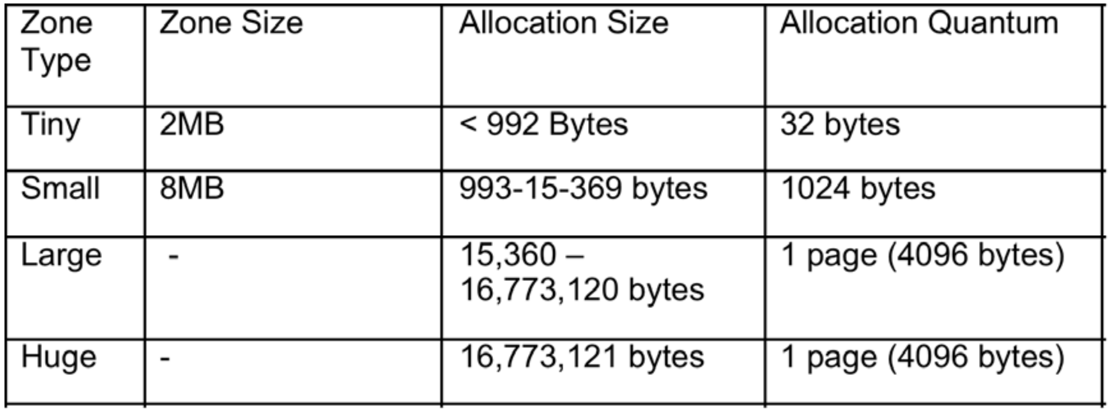
Ruby-DTrace and the Heap

- Gera’s Heap Tracer works with truss or ltrace to hook the functions that make up the heap (malloc(), calloc(), free(), mmap()), etc.
- ImmLib does the same
- We can do this with RE:Trace and more
- Not only can we hook dynamic allocation functions to “watch” the heap
- We can also determine what functions are writing to the heap and hook their arguments to find heap overflows, double free(), and double malloc()

RE:Trace Heap Smasher()

Refresher:

- When you malloc() on OS X, you are actually calling the scalable zone allocator, which breaks allocations into different zones by size:



Zone Type	Zone Size	Allocation Size	Allocation Quantum
Tiny	2MB	< 992 Bytes	32 bytes
Small	8MB	993-15-369 bytes	1024 bytes
Large	-	15,360 – 16,773,120 bytes	1 page (4096 bytes)
Huge	-	16,773,121 bytes	1 page (4096 bytes)

Adapted from: *OS X Internals A System Approach*

RE:Trace Heap Smasher()

- In our heap smash detector, we must keep track of four different heaps
- We do this by hooking malloc() calls and storing them to ruby hashes with the pointer as the key and the size allocated as the value
- We break the hashes into tiny, small, large, and huge by allocation size
- We then hook all allocations and determine if the pointer falls in the range of the previous allocations. We can adjust the heap as memory is free()'d or realloc'd()

RE:Trace Heap Smasher()

- In the process, we can detect double free()'s double malloc's leak, etc.
- There are similar tools to do this already (malloc debug, memory leak tools), but DTrace can be tailored to the application you are researching
- We can easily tailor our output to work with Gera's Heap Tracer OpenGL interface or write our own with Ruby-OpenGL
- The really interesting functionality is used to look for errors in malloc usage

RE:Trace Heap Smasher()

- By hooking C functions(strncpy, memcpy, memmove, etc.) we can determine if they are overallocating to locations in the heap by looking at the arguments

```
pid$target::strncpy:entry {
   self->sizer = arg2;
   printf("copyentry:dst=0x%p|src=0x%p;size=%i", (int) *(int *)
             copyin(arg0, 8), (int) *(int *) copyin( (user_addr_t) arg1, 4), arg2);
   self->sizer = 0;
}
```

RE:Trace Heap Smasher()

- We can check to see if the allocation happens in a range we know about (check the hash). If it does, we know the size allocation, and we can tell if a smash will occur
- Compared to our stack smash detector, we need very few probes. A few dozen probes will hook all the functions we need
- We can attach to a live process on and off without disturbing it
- It is better to start the process with heap smash so we don’t miss anything

RE:Trace Heap Smasher()

- We also keep a hash with the stack frame, which is called the original malloc()
- When an overflow is detected, we know:
  - Who allocated it (stack frame)
  - Who used it (function hook)
  - Where the overflowed memory is
  - How large the overflow was

SAIC. From Science to Solutions

RE:Trace Heap Smasher()

```
hochi@TEKDBZ:~$ sudo ruby heapsmash.rb 938
initializing probes...
starting tracing...
HEAP OVERFLOW DETECTED!!! AT ADDRESS
    0x16e6c000
BY PROBE: pid938:libSystem.B.dylib:memcpy:entry
    copyentry:dst=0x16e6c000|src=0x7269662f;size=56
DEST SIZE 48
COPY SIZE 56
MALLOC'D FROM: libmozjs.dylib`JS_malloc+0x1d
```

RE:Trace Heap Smasher()

- Future additions:
- Graphviz/OpenGL Graphs
- There is a new version of Firefox which has probes in the JavaScript library
- This would give us functionality similar to Alexander Soitorov's HeapLib (Heap Fung Shui) for heap manipulation generically
- Safari should follow soon
- You tell me?

Using DTrace Defensively

DTRACE DEFENSE

Basic HIDS with DTrace

- Return to LibC, the major attack vector for stack-based buffer overflows
- Return to Mprotect() is also big
- Using Dtrace, you can profile your applications basic behavior
- You should then be able to trace for anomalies with predicates
- This is great for hacking up something to protect a custom application
- Easy to create a rails interface with Ruby-DTrace

Basic HIDS with DTrace

- Problem: “I want to use QuickTime, but it’s got more holes than something with a lot of holes”
- Make a DTrace script to call stop() when weird stuff happens
- QuickTime probably never needs to call /bin/sh or mprotect() on the stack to make it readable (Houston we have a problem)
- Then again…

Basic HIDS with DTrace

```
#!/usr/sbin/dtrace  -q -s

proc:::exec
     /execname == "QuickTime Playe" &&
     args[0] == "/bin/sh"/
{
     printf("\n%s Has been p0wned! It tried
to  spawned %s\n”, execname, args[0])
}
```

DTrace and Rootkits

- Check out Archim’s paper “B.D.S.M the Solaris 10 Way,” from the CCC Conference
- He created the SInAr rootkit for Solaris 10
- Describes a method for hiding a rootkit from DTrace
- Only works on SPARC
- DTrace FBT (kernel) provider can spy on all active kernel modules
- Should have the ability to detect rootkits, which don’t explicitly hide from DTrace (SInAr is the only one I could find)
- Expect more on this in the future

DTrace for Malware Analysis

- Very easy to hack up a script
- Recent Leopard affected DNS Changer (**OSX.RSPlug.A)**
- Why the heck is my video codec calling…
- /usr/sbin/scutil
- add ServerAddresses * $s1 $s2
- set State:/Network/Service/$PSID/DNS
- You can monitor file I/O and syscalls with just two lines
- Scripts to do this now included with OS X by default
- Malware not hiding from DTrace yet
- BUT Apple made that a feature (yayyy!)

Hiding from DTrace

- Core DTrace developer Adam Leventhal discovered that Apple crippled DTrace for Leopard
- This is against the basis for DTrace
- Your application can set the “PT_ATTACH_DENY” flag to hide from DTrace just like you can for GDB
- Leventhal used timing attacks to figure out they are hiding iTunes™ and QuickTime from DTrace
- http://blogs.sun.com/ahl/entry/mac_os_x_and_the
- Very easy to patch in memory or with kext
- Landon Fuller released a kext to do this
- http://landonf.bikemonkey.org/code/macosx/Leopard_PT_DENY_ATTACH.20080122.html

Conclusion

DTrace can:

- Collect an unprecedented range of data
- Collect very specific measurements
- Scope can be **very broad** or **very precise**

Applied to Reverse Engineering:

- Allows researchers to pinpoint specific situation (overflows)
- Or to understand general behavior (heap growth)

See the RE:Trace framework for implementation

Future Work

- Automated feedback and integration with fuzzers
- Kernel tracing
- Improved overflow monitoring
- Utilizing application-specific probes (probes for JS in browsers, MySQL probes, ...)

Your own ideas!

Thank You!

Questions?

Tiller Beauchamp
SAIC
Tiller.L.Beauchamp@SAIC.com

David Weston
SAIC
David.G.Weston@saic.com