ENDGAME.

Sense. Discover. Act.™

IDA Splode

Dynamically Enhancing Static Analysis

IDA Splode - Overview

- Background
- Motivation
- Walkthrough
- Examples
- Future
- Demo?

Background

IDA Splode - Background

ENDGAME.

IDA-Splode

- Horrible name for the tool I wrote, which combines…
- Hex-Rays IDA Pro
  - Application to disassemble software
  - **Static** analysis
  - `while(1){}` → `0xebfe: jmp $-2`
- IDA Python
  - Python bindings for IDA's API
  - `while ea != BADADDR: ea = NextHead(ea)`
- Intel's Pin
  - Dynamic binary instrumentation framework
  - Effectively hook executing code at the instruction level
  - `INS_AddInstrumentFunction(CalledOnEachInstruction)`

IDA Splode - Background

ENDGAME.

Pin and Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI) is a method of analyzing the behavior of a binary application at runtime through the injection of instrumentation code. This instrumentation code executes as part of the normal instruction stream after being injected.

– Skape (Uninformed v7a1)

Motivation

IDA Splode - Motivation

ENDGAME.

Origins of IDA Splode

- Single-byte mutation
  - Heap overflow with corrupted data
  - Decompression failure
- Root Cause Analysis of Heap Overflow
  - Parser for undocumented format
  - Closed source, but have debugging symbols
  - All file data is compressed (flat entropy)
  - Extensive use of COM interfaces
  - Lots of C++ isms (OO, virtual inheritance, etc)
  - Lots of shiny things to get distracted by
- Too much churning, need better tools

Motivation for Developing IDA Splode

- IDAPython
  - Limited exposure
  - Small-scale applications
- Pin
  - Powerful tool with lots of potential
  - No previous personal experience
  - Some previous organizational experience
- Exists to answer
  - Given `mov [esi+0x18], eax`
  - What is in the memory `esi` points to?
  - Where did the allocation come from?
  - Which routines touch that data?

IDA Splode - Motivation

Static Reversing & Runtime Analysis

- Load binary in Windbg
  - Breakpoint on instruction
  - `!heap, kv, ln, dd, wt`
  - All require useful symbols, standard allocators
- Load binary in IDA
  - Annotate nearby instances of `esi`
  - Maybe create a struct
  - Tedious, manual, error-prone
- Single-step sample size n=1, no automation

IDA Splode - Motivation

ENDGAME.

Dynamically Enhanced Analysis

- PIN Tool + IDAPython
- Log all memory accesses
  - Instrument all `mov/movzx/lea/cmp`
- Record metadata at runtime
  - Stack?
    - Instrument all `call` and `ret` and instructions
    - Given stack pointer, know parent frame & BP offset
  - Heap?
    - Page Heap gives allocation size and stack trace
    - Given heap pointer, know size, offset and origin
  - Symbol?
    - Windbg/DbgHelp.dll to resolve symbols

IDA Splode - Motivation

ENDGAME.

Scope and Features

- Scope Limited by Pin's OS Support
  - Intel CPUs only (Windows, Linux, OSX, Android-on-Intel)
    - Currently Windows-only
  - For other OSes, would need to change
    - Debug symbol resolution (macho, elf)
    - Heap allocation tagging
- Does
  - Augment reversing (no source)
  - Make getting started easier
  - Get more useful with debug symbols
- Does Not
  - Detect OOB access or exploitable conditions (a la Valgrind)
  - Track taints, symbolic, concolic, etc. analysis
  - If you see "symbol" or "symbolic" in this presentation, think Windbg

Basic Use Walkthrough

IDA Splode - Walkthrough

ENDGAME.

Three Parts to IDA Splode

- Running Pin Instrumentation
- Import Traces Into Database
- IDA Pro Script, Usage, and Hotkeys

Running Pin Instrumentation

- Acquire `ida-splode.dll` pintool (on SVN)
- Run target binary under Pin, specify ida-splode.dll tool
  - `pin.exe -t ida-splode.dll -- target.exe args`
- Outside scope of presentation, knobs available for…
  - Metadata gathering
    - Heap vs. Stack vs. Debug Symbols
  - Limiting scope of instrumentation or logging
    - By module (-m foo.dll) or routine (-r foo!Function)
  - Types of instructions instrumented
    - MOV vs. LEA vs. CMP
    - Instrument registers vs. const values

ENDGAME.

Import Traces into Database

- Uses MongoDB for ease-of-use and zero configuration
    - `\path\to\bin\mongod --dbpath "any folder"`
- Log files created by Pin instrumentation
    - All logs are plain Python
        - One dict per instrumented instruction
        - Greppable
    - Run target.exe.py to import traces into MongoDB
        - Traces are in target.exe.py.traces.py
        - Modules are in target.exe.py.modules.py

IDA Splode - Walkthrough

ENDGAME.

IDA Pro Script & Usage

- Single script to run via ALT+F7 or “File>Script File”
  - `ida-splode\py\idapython_script.py`
- Upon execution
  - Displays usage and hotkeys
  - Connects to MongoDB on localhost
  - Shows a list of candidate databases
    - Must contain a module with matching MD5
  - First run, asks user which database to load
  - Subsequent runs, auto-loads previously-used database
- Common Hotkeys
  - ^⇧H   Help and usage
  - ^⇧C   Color instructions in module
  - ^⇧S   Summary for selected instruction
  - ^⇧D   Details for selected instruction
  - ^⇧X   Create X-refs from module trace info
  - ^⇧L   Load a different MongoDB database

Examples & Screen Shots

IDA Splode - Examples

ENDGAME.

Example Usage Cases

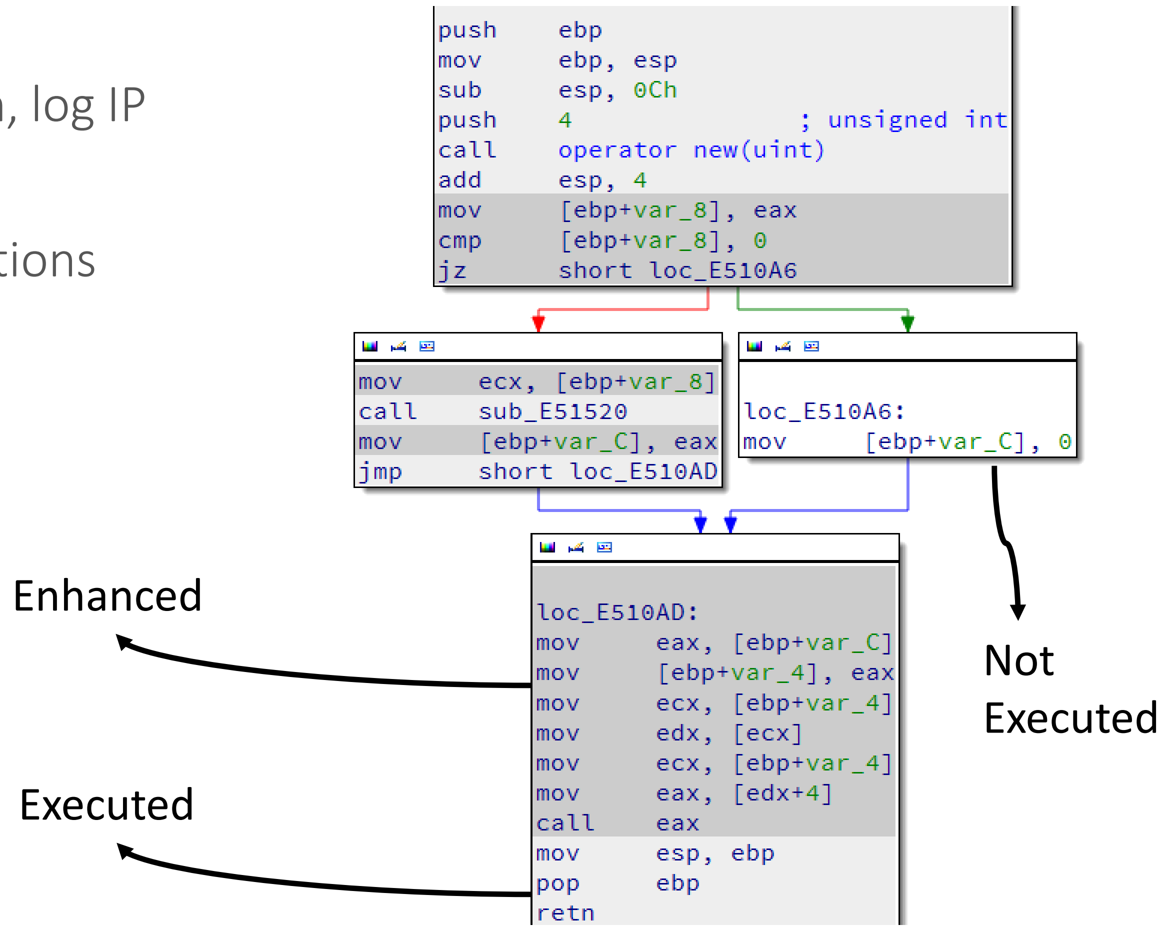
- Easiest to demonstrate in bite-sized, contrived examples
- Features Demonstrated
  - Instruction Tracing
  - Memory Value Recording
  - Branch Tracing & Statistics
  - Vtable Resolution
  - IAT Rebuilding
  - Extramodular xrefs
  - Branch Statistics
  - Stack Variable Propagation
  - Heap Metadata

IDA Splode - Examples

Instruction Tracing

- Simplest PIN task
  - On each instruction, log IP
- Simplest IDA task
  - Color those instructions

Example Disassembly



IDA Splode - Examples

ENDGAME.

Memory Values

- For most instructions…
  - Log memory value read/set
  - Log memory address
- Basic stats on common vals
- Values stand out

Example Disassembly

- Normal Run

```
mov     [ebp+var_14], ecx ; >< 24h
```

- Crashing Run

```
mov     [ebp+var_14], ecx ; >< 8080A47Ch(50%), 24h(50%)
```

IDA Splode - Examples

ENDGAME.

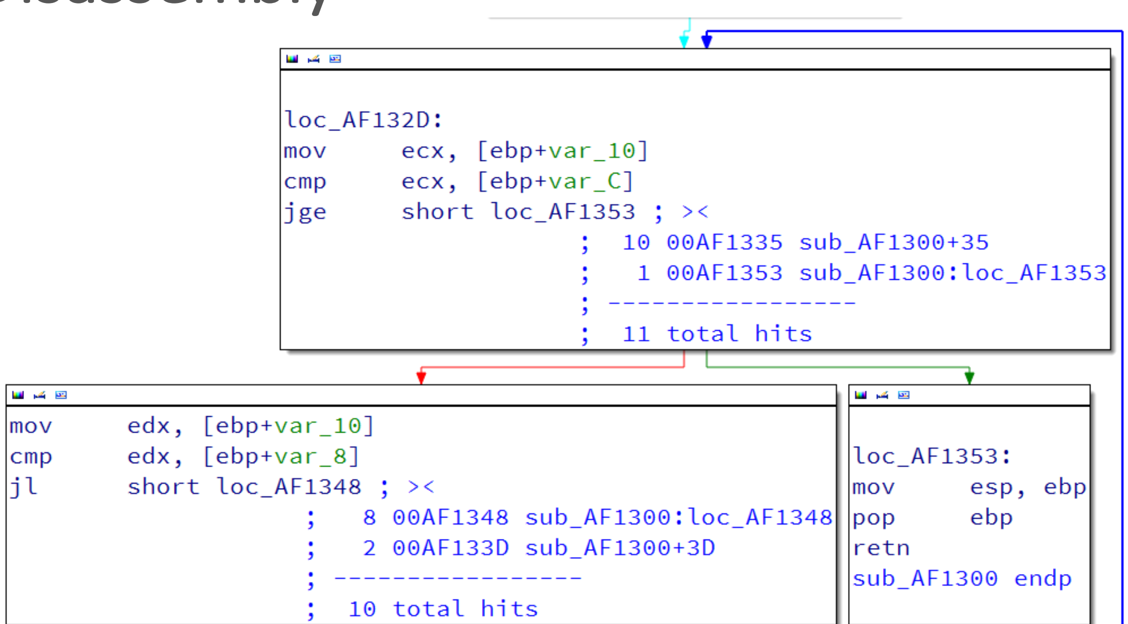
Branch Tracing

- Logs branches taken
- Generates statistics
- Useful for
  - Loops
  - Error handlers

Example Source

```
void TestBranchStatistics() {
    int x = 0, ten=10, eight=8;
    for(int i = 0; i < ten; i++)
    {
        if(i >= eight) { x++; }
        else           { x--; }
    }
}
```

Example Disassembly



IDA Splode - Examples

ENDGAME.

Vtable Call

- Lots of Pin Tools do this
  - Might as well do it too
- Adds XREFs

Example Source

```
struct V {
    virtual ~V(){};
    virtual void Foo()=0;
};
struct V1 : public V { void Foo(){} };
void TestVtable() {
    V *v = new V1;
    v->Foo();
}
```

Example Disassembly

```
mov     ecx, [ebp+v]
mov     eax, [edx+4]
call    eax ; ><   1 00E51540 V1::Foo
```

IDA Splode - Examples

ENDGAME.

Dynamic Imports

- Lots of Pin Tools do this
  - Might as well do it too
- Dynamic Import Section
  - Adds function stub
  - Adds XREFs

Example Source

```
typedef void (WINAPI * PfnODS)(char*);
PfnODS pODS = NULL;
#define szMod      "kernel32.dll"
#define szFnName   "OutputDebugStringA"
void TestDynamicIAT() {
    HMODULE h = GetModuleHandleA(szMod);
    pODS = (PfnODS) GetProcAddress(h, szFnName);
    pODS("0x1239");
}
```

Example Disassembly

```
call    dword_B8345C    ; ><   1 00B94000 KERNEL32!OutputDebugStringA
```

IDA Splode - Examples

ENDGAME.

Extramodular Xrefs

- Dynamic imports is only half the battle
- Where else is <API> called?
- Segment created for external refs
  - Create Fn stub for each enhanced insn
  - …or batch mode (entire module)

Example Disassembly

```
.idasplode:00B94000 KERNEL32_OutputDebugStringA: ; CODE XREF: sub_B81290+2B↑p
.idasplode:00B94000         retn    4
```

IDA Splode - Examples

ENDGAME.

Stack Tracking

- Tracks all stack frames
  - Hook each *call/ret*
  - Save ESP after *call*
  - Check ESP after *ret*
- Ptr-to-stack = obvious

Example Source

```
void stack_var_user(int* C) {
    *C = 0x1237;
}
void stack_var_middleman(int* B) {
    stack_var_user(B);
}
void TestStackVars() {
    int A = 0x1238;
    stack_var_middleman(&A);
}
```

Example Disassembly

```
mov     eax, [ebp+C] ; >< 00D41260 TestStackVars A
mov     dword ptr [eax], 1237h
```

IDA Splode - Examples

ENDGAME.

Heap Metadata

- Contrived Example
  - Create struct on heap
  - Set values inside struct
  - __thiscall and __stdcall

Example Source

```
#include <pshpack1.h>
struct S {
    char  a;
    short b;
    int   c;
    S() { c = 0x1230;
          b = 0x1231;
          a = 1; }
    void setA() { a = 1; }
    void setB() { b = 0x1232; }
    void setC() { c = 0x1233; }
};
#include <poppack.h>

void UseS(S* s) { s->c = 0x1234; }

void TestStruct() {
    S *s = new S();
    s->setA();
    s->setB();
    s->setC();

    UseS(s);

    delete s;
}
```

IDA Splode - Examples

ENDGAME.

Heap Metadata

- var_8 looks like a object
  - Where is it from?
  - How big is it?
  - Where else is it used?
- How can IDA Splode help?

Example Disassembly

```
mov     ecx, [ebp+var_8]
call    sub_B81490
mov     [ebp+var_10], eax
jmp     short loc_B8103D
```

IDA Splode - Examples

ENDGAME.

Summary View

- “Value” only
- Single hotkey to enhance
- Smaller amount of screen space

Example Disassembly

Allocation size in bytes

```
mov     ecx, [ebp+var_8] ; ><
                         ; 0 bytes into 7-byte alloc from
                         ; 00E51018 sub_E51010+8
```

Offset Into Allocation
(ecx is base of alloc)

First useful frame of
alloc stack trace

IDA Splode - Examples

ENDGAME.

Details View

- “Address” & “Value”
  - Does what it says on tin
- Heap metadata
  - More exploded view
  - Statistics & Stack Trace

Example Disassembly

```
mov     ecx, [ebp+var_8] ; ><
                 ; ===== Address =====
                 ; => Stack Pointers
                 ;     00E51010 sub_E51010 var_8
                 ; ===== Value =====
                 ; => Heap Pointers
                 ; Count  Pctg  Offset Size
                 ;     1  100%     0x0  0x7
                 ; Backtrace:
                 ;     00E51018 sub_E51010+8
                 ;     00E5142B _main+1B
call    sub_E51490
```

Where are we reading from or writing to?

What was read from or written to that address?

IDA Splode - Examples

Details View

- “Address” & “Value”
  - Does what it says on tin
- Heap metadata
  - More exploded view
  - Statistics & Stack Trace

Example Disassembly

```
mov     ecx, [ebp+var_8] ; ><
                         ; ===== Address =====
                         ; => Stack Pointers
                         ;     00E51010 sub_E51010 var_8
                         ; ===== Value =====
                         ; => Heap Pointers
                         ; Count  Pctg  Offset Size
                         ;     1  100%     0x0 0x7
                         ; Backtrace:
                         ;     00E51018 sub_E51010+8
                         ;     00E5142B _main+1B
call    sub_E51490
```

Tells us what we already know.

Heap!
One distinct allocation size/offset pair, which is seen 100% of the time.

Allocation stack trace; trims off actual allocator name (e.g. malloc/new). Addresses clickable.

IDA Splode - Examples

ENDGAME.

Heap Metadata

- var_8 looks like a object
- Where is it from?
  - Created at sub_E51010+8
  - Easy search forward for c'tor
- How big is it?
  - 7 bytes
- What does it look like?
- Where else is it used?

```
mov     ecx, [ebp+var_8] ; ><
                         ; 0 bytes into 7-byte alloc from
                         ; 00E51018 sub_E51010+8
```

Summary Hotkey

```
mov     ecx, [ebp+var_8]
call    sub_E51490
```

Details Hotkey

```
mov     ecx, [ebp+var_8] ; ><
                    ; ===== Address =====
                    ; => Stack Pointers
                    ;     00E51010 sub_E51010 var_8
                    ; ===== Value =====
                    ; => Heap Pointers
                    ; Count  Pctg  Offset Size
                    ;     1  100%     0x0 0x7
                    ; Backtrace:
                    ;     00E51018 sub_E51010+8
                    ;     00E5142B _main+1B
call    sub_E51490
```

IDA Splode - Examples

ENDGAME.

Heap Metadata

- var_8 looks like a object
- Where is it from?
  - Created at sub_E51010+8
  - Easy search forward for c'tor
- How big is it?
  - 7 bytes
- What does it look like?
  - Three fields (1b, 2b, 4b)
- Where else is it used?
  - sub_E51490
  - sub_E514C0
  - Etc.

```
struct S {
    char  a;
    short b;
    int   c;
```

Compile

```
mov     ecx, [ebp+var_8]
call    sub_E51490
```

Reconstruct Hotkey

```
00000000 MyStruct struc ; (sizeof=0x7)
00000000 autofield_0 db ?    ; XREF: sub_E51490+20↑w sub_E514C0+A↑w
00000001 autofield_1 dw ?    ; XREF: sub_E51490+19↑w sub_E514E0+F↑w
00000003 autofield_3 dd ?    ; XREF: sub_E51000+6↑w sub_E51490+A↑w ...
00000007 MyStruct ends
```

IDA Splode - Examples

ENDGAME.

Vtable Call (Again)

- Same Example
- Heap metadata enhanced
  - Provides a bigger picture

Source

```
struct V {
    virtual ~V(){};
    virtual void Foo()=0;
};
struct V1 : public V { void Foo(){} };
void TestVtable() {
    V *v = new V1;
    v->Foo();
}
```

```
mov     ecx, [ebp+v] ; ><
                ; 0 bytes into 4-byte alloc from
                ; 00E51088 TestVtable+8
mov     eax, [edx+4] ; ><
                ; ===== Address =====
                ; => Symbols
                ;     00E5217C const V1::`vftable'+4
                ; ===== Value =====
                ; => Symbols
                ;     00E51540 V1::Foo
call    eax ; ><   1 00E51540 V1::Foo
```

IDA Splode - Examples

ENDGAME.

Scenario: Custom Allocator

- Can't, or won't, change to PageHeap-compatible
  - E.g. Closed-source software implementing its own heap
- Redirect execution at runtime
- In this example, *custom_malloc* redirected to *malloc*

```
int heap[0x100];
void* custom_malloc(int size) { return &heap[0x80]; }
```

```
call    custom_malloc(int)
add     esp, 4
mov     [ebp+memory], eax ; ><
                ; ===== Address =====
                ; => Stack Pointers
                ;     00E511B0 TestCustomMalloc memory
                ; ===== Value =====
                ; => Heap Pointers
                ; Count  Pctg  Offset Size
                ;     1  100%     0x0 0x1236
                ; Backtrace:
                ;     00E511E8 TestCustomMalloc+38
                ;     00E5144E _main+3E
```

Works because of redirection to *malloc*

IDA Splode - Examples

ENDGAME.

Scenario: Custom Allocator

- Can’t, or won’t, change to PageHeap-compatible
  - E.g. Closed-source software implementing its own heap
- Redirect execution at runtime
- In this example, *custom_malloc* redirected to *malloc*

```
// -- Demo application allocator hooks
new SymbolResolver<INS>(new MallocPageHeapRedirector(0,1), "demo.exe!custom_malloc");
```

IDA Splode - Examples

ENDGAME.

Moveable Heap

- GlobalAlloc & LocalAlloc
  - Allow MEM_MOVEABLE flag
  - Used by IStream COM iface
  - E.g. lots of MSFT code
- Get handle to allocation
  - OS can move around allocation & data unless “Locked”
- No PageHeap Support
  - Hook RtlAllocateHandle
  - Keep track at runtime
  - HGLOBALs are heap pointers, not possible to confuse a la HANDLEs

```
void TestMoveableHGLOBAL() {
    HGLOBAL g = GlobalAlloc(GHND, 0x123a);
    void *  m = GlobalLock(g);
    GlobalUnlock(m);
    GlobalFree(g);
}
```

```
push    123Ah    ; dwBytes
push    GMEM_MOVEABLE or GMEM_ZEROINIT ; uFlags
call    ds:GlobalAlloc
mov     [ebp+hMem], eax ; ><
                 ; 0 bytes into 123Ah-byte alloc from
                 ; 00E512DD sub_E512D0+D
```

Future

IDA Splode - Future

ENDGAME.

Potential Enhancements & Future

- Scalability & Analytics
  - Lots of trace data generated every run
  - 80mb of data for ~0.030 seconds of computation
- Defense
  - CTFGRIND + EMET emulation
- Valgrind Rewrite
  - Useful for ARM binaries
- Linux/OS X Support
  - Not everything is Windows-based
- More intuitive/useful UI
  - Comments are quick-and-easy
  - IDA runs a full QT UI

Questions?

ENDGAME.

Sense. Discover. Act.™