

# Automation Techniques in C++ Reverse Engineering

Rolf Rolles, Möbius Strip Reverse Engineering

August 5, 2019

## Introduction

### Dynamic Structure Reconstruction

- Existing DBI-Based Approaches
- Limitations of DBI-Based Solutions
- My Contributions to this Problem

### Dynamic Resolution of Argument Types

- Preprocessing
- Run-Time Data Collection
- Applying the Results

### Further Extensions and Challenges

- Extensions
- Challenges

## Conclusion

# Introduction

## Genesis of this Research

- ▶ While researching an upcoming C++ RE training class, I:
  - ▶ Practiced statically reverse engineering large C++ binaries.
  - ▶ Spent ~85%-95% of my time creating and setting types.
  - ▶ Experimented with automating type-related activities.
    - ▶ A few of my results are detailed in this presentation.
- ▶ **Goal:** derive type-related metadata from runtime allocation and structure access data, and apply it in IDA and Hex-Rays.
  - ▶ The techniques are simple, but the results are very useful!
- ▶ Two primary analyses, both based on DLL injection:
  1. Track structure accesses
  2. Track data flow from allocation sites into function arguments

# Introduction

## Type Information

```
__int64 __fastcall sub_17142D60(__int64 a1, _DWORD *a2)
{
    unsigned __int8 *v3; // rbp
    __int64 v4; // rsi
    __int64 result; // rax

    if ( a2 != (_DWORD *)a1 )
    {
        v3 = (unsigned __int8 *) (a2 + 8);
        v4 = a1 + 32;
        if ( a2 + 8 != (_DWORD *) (a1 + 32) )
        {
            sub_17144EB0((unsigned __int8 *) (a1 + 32));
            sub_17142E10(v4, v3);
        }
        if ( a2 + 12 != (_DWORD *) (a1 + 48) )
        {
            sub_17144EB0((unsigned __int8 *) (a1 + 48));
            sub_17142E10(a1 + 48, (unsigned __int8 *) a2 + 48);
        }
        if ( a2 + 16 != (_DWORD *) (a1 + 64) )
        {
            sub_17144EB0((unsigned __int8 *) (a1 + 64));
            sub_17142E10(a1 + 64, (unsigned __int8 *) a2 + 64);
        }
        *(_DWORD *) (a1 + 24) = a2[6];
        *(_DWORD *) a1 = *a2;
        result = (unsigned int) a2[1];
        *(_DWORD *) (a1 + 4) = result;
    }
    return result;
}
```

Type information is the difference between this: unreadable, borderline useless gibberish ...

# Introduction

## Type Information

```
void __fastcall sub_17142D60(minsn_t *a1, minsn_t *a2)
{
    mop_t *v3; // rbp
    mop_t *v4; // rsi

    if ( a2 != a1 )
    {
        v3 = &a2->l;
        v4 = &a1->l;
        if ( &a2->l != &a1->l )
        {
            sub_17144EB0(&a1->l);
            sub_17142E10(v4, v3);
        }
        if ( &a2->r != &a1->r )
        {
            sub_17144EB0(&a1->r);
            sub_17142E10(&a1->r, &a2->r);
        }
        if ( &a2->d != &a1->d )
        {
            sub_17144EB0(&a1->d);
            sub_17142E10(&a1->d, &a2->d);
        }
        a1->ea = a2->ea;
        a1->opcode = a2->opcode;
        a1->iprops = a2->iprops;
    }
}
```

... and this: nearly perfect code versus the original source, minus names and comments. However, it is tedious to create and apply type information, so let's automate it.

# Introduction

## Interesting Type-Related Information

Discover, through dynamically executing the program:

- ▶ All exercised allocation sites and their sizes
- ▶ Size and layout of structures; sizes for its fields
  - ▶ Also discover structures contained within other structures
- ▶ All locations accessing allocated structures of interest
- ▶ Type relationships between fields of different structures
- ▶ Function argument and local variable types

(And, some more experimental stuff described later.)

# Introduction

## Numbers for my Current Target

These techniques allowed me to automatically (or semi-auto.) create and apply type information for my current target:

Structures recovered	~200
Structure references added	10,000+
Union selections applied	~2,200
Variable types modified	~6,000
Argument types modified	~2,750

## Dynamic Structure Reconstruction

### Existing DBI-Based Approaches

- Locate Memory Management Functions
- Hook Memory Management Functions
- Run the Program, Instrumented
- Instrument Memory References
- Detect and Record Structure Accesses
- Post-Process Recorded Data

### Limitations of DBI-Based Solutions

### My Contributions to this Problem



# Dynamic Structure Reconstruction

## Inspiration

- ▶ Existing academic work on the subject inspired me:
  - ▶ Howard: A Dynamic Excavator for Reverse Engineering Data Structures by Slowinska et al
  - ▶ dynStruct: An Automatic Reverse Engineering Tool for Structure Recovery and Memory Use Analysis by Mercier
    - ▶ The author has published the source code on GitHub.
- ▶ I adapted and modified their ideas for better performance and increased flexibility.
  - ▶ For example, I use DLL injection instead of DBI.

# Dynamic Structure Reconstruction

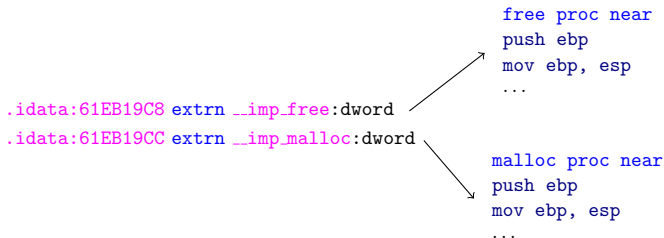
## Overview

The workflow of these tools is as follows:

1. Locate addresses of `malloc`, `free`, etc.
2. Hook these memory routines at runtime to record:
  - ▶ The allocation site (e.g., address of the call to `malloc`)
  - ▶ The size of the allocation
  - ▶ The pointer returned by `malloc`
  - ▶ Discard this information upon a call to `free`
3. Run the program under dynamic binary instrumentation (DBI).
4. **Instrument every instruction that accesses memory.**
5. Upon memory access, if address is within an allocation, log:
  - ▶ Address of referencing instruction
  - ▶ Allocation details (allocation site and size)
  - ▶ Accessed offset within allocation
6. Post-process the data to build higher-level information.

# Dynamic Structure Reconstruction

## Step #1: Locate Memory Management Functions



Locate and record pointers to memory management functions.

(Of course, these may be contained in the binary and require direct hooks.)

This step is not specific to DBI.

# Dynamic Structure Reconstruction

## Step #2: Hook Memory Management Functions

```
.idata:61EB19C8 extrn __imp_free:dword  —HOOK→ &freeHook  
.idata:61EB19CC extrn __imp_malloc:dword —————→ &mallocHook  
HOOK
```

Hook the routines, point them to our wrappers around them (somewhere inside of the same address space).

This step is not specific to DBI.

# Dynamic Structure Reconstruction

## Skeletons for the Memory Management Wrappers

The hooks save metadata upon `malloc`, and discard upon `free`.

```
void *mallocHook(int size) {  
#1 void *mem = pfnOriginalMalloc(size);  
#2 remember(mem, size, _ReturnAddress());  
#3 return mem;  
}
```

1. Invoke the original `malloc`
2. Record the pointer / size / allocation site
3. Return the allocated pointer

```
void freeHook(void *mem) {  
#1 forget(mem);  
#2 pfnOriginalFree(mem);  
}
```

1. Remove metadata about that allocation
2. `free` it

This works transparently to unmodified applications.

# Dynamic Structure Reconstruction

## `remember` and Allocation Records

`remember` stores **allocation records**.

### Allocation Record

Allocated pointer	Size	RVA of return address from <code>malloc</code>
-------------------	------	--

---

The allocation site is written as an RVA (offset into image).

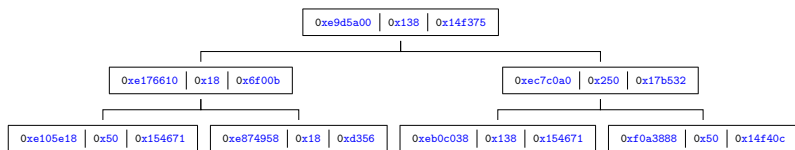
```
.text:61EC53EE  call  malloc
.text:61EC53F3  mov   rbx, rax ◀
```

Base address is `61EB0000`, so RVA ◀ = `61EC53F3-61EB0000` = `153F3`.

# Dynamic Structure Reconstruction

Implementation of `remember` and `forget`

`remember` stores pointers and metadata in a tree (map) structure.  
`forget` removes items from the tree.



Data should be kept sorted so we can lookup addresses *within* allocations.  
Binary trees (AVL, red/black) are well-suited; hash tables generally aren't.

## Dynamic Structure Reconstruction

### Existing DBI-Based Approaches

Locate Memory Management Functions

Hook Memory Management Functions

**Run the Program, Instrumented**

Instrument Memory References

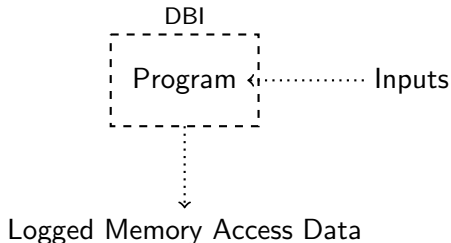
Detect and Record Structure Accesses

Post-Process Recorded Data



# Dynamic Structure Reconstruction

## Step #3: Run Program under Instrumentation

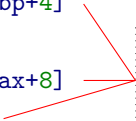


Run the program under dynamic binary instrumentation. Provide inputs that exercise as much functionality as possible.

# Dynamic Structure Reconstruction

## Step #4: Instrument Memory References

```
.text:61F33E5A mov eax, [ebp+4]
.text:61F33E60 add edx, 4
.text:61F33E63 mov ebx, [eax+8]
.text:61F33E69 push ebx
.text:61F33E6A call sub_61F33950
```



Insert DBI memory access  
callback routine before  
**every** memory access

Use DBI to instrument **every** memory reference.

# Dynamic Structure Reconstruction

## Step #5: Detect, Record Structure Accesses

```
void DBIMemAccessCallback(  
    ADDR eaIns, ADDR eaMem,  
    SIZE size, BOOL bRead) {  
    {  
        // Lookup address in map  
        AllocRecord *ar = lookup(eaMem); ◀  
        // Was it within an allocation?  
        if(ar != NULL)  
            log(ar, eaIns, size, eaMem, bRead); ◀  
    }  
}
```

- ▶ Lookup accessed addresses in the map. ◀
- ▶ If the access was within an allocation, log the details. ◀
  - ▶ Next slide gives an example.

# Dynamic Structure Reconstruction

## Step #5: Detect, Record Structure Accesses

Suppose this instruction accesses address **DC07928**:

```
.text:61F33E63 mov ebx, [eax+8]
```

Suppose further that we have recorded this allocation record:

<b>DC07900</b>	<b>80</b>	<b>6F00B</b>
----------------	-----------	--------------

We found an access! Log the following data:

Allocation RVA	<b>6F00B</b>
Allocation Size	<b>80</b>
Instruction RVA	<b>83E63</b>
Access Size	<b>dword</b>
Access Offset	<b>0x28</b>
Access Type	<b>READ</b>

## Dynamic Structure Reconstruction

### Existing DBI-Based Approaches

Locate Memory Management Functions

Hook Memory Management Functions

Run the Program, Instrumented

Instrument Memory References

Detect and Record Structure Accesses

Post-Process Recorded Data

# Dynamic Structure Reconstruction

## Step #6: Post-Process Recorded Data

So far, we have logged access data to allocated objects, as in:

Allocation		Inst.	Access		
RVA	Size	RVA	Offset	Size	Type
6F00B	50	6D04A	0x48	4	WRITE
14F40C	50	6B84D	0x18	4	READ
14C213	50	6B859	0x4	4	READ
55816	50	1E0E4	0x44	4	READ
E941A	50	BD7DC	0x10	8	WRITE
6F00B	50	6D000	0x8	8	READ
55816	50	6D00B	0x0	4	WRITE
6F00B	50	149E8D	0x20	1	READ

Now we process this data to reconstruct useful information.

This step is not specific to DBI.

# Dynamic Structure Reconstruction

## Segregate Data by Allocation Site

	Allocation		Inst.	Access		
	RVA	Size	RVA	Offset	Size	Type
▶	6F00B	50	6D04A	0x48	4	WRITE
	14F40C	50	6B84D	0x18	4	READ
	14C213	50	6B859	0x4	4	READ
▶	55816	50	1E0E4	0x44	4	READ
	E941A	50	BD7DC	0x10	8	WRITE
▶	6F00B	50	6D000	0x8	8	READ
▶	55816	50	6D00B	0x0	4	WRITE
▶	6F00B	50	149E8D	0x20	1	READ

First, group accesses by their allocation site.

(If two sites are known to allocate the same type, we can merge their data.)

# Dynamic Structure Reconstruction

## Rebuild C-Level Structures

For a given allocation site:

Offset	Size	
0x0	4	
0x4	4	
0x8	8	
0x10	8	
0x18	4	
0x20	1	
0x21	1	
0x22	2	
	...	

1. Sort accesses, remove duplicates.



# Dynamic Structure Reconstruction

## Rebuild C-Level Structures

For a given allocation site:

Offset	Size	struct X {
0x0	4	int f0;
0x4	4	int f4;
0x8	8	__int64 f8;
0x10	8	__int64 f10;
0x18	4	int f18;
0x20	1	char f20;
0x21	1	char f21;
0x22	2	short f22;
		...

1. Sort accesses, remove duplicates.
2. Create properly sized and padded fields. (Easy, right? ...)

# Dynamic Structure Reconstruction

## Discover Nested Subobjects

Brief digression: discovery of nested structure locations.

```
mov ebx, [eax+▶8◀]
```

Allocation RVA	6F00B
Allocation Size	80
Access Offset	▶ 0x28 ◀

- ▶ Notice that the instruction vs. accessed offsets are different:
  - ▶ The instruction uses offset ▶8◀; however:
  - ▶ The logged, raw offset into the allocation was ▶ 0x28 ◀.
- ▶ Hence, when logged, `eax` pointed +0x20 into the allocation.
  - ▶ *Usually*, this implies a structure is contained at offset +0x20.
  - ▶ The alternative is a pointer to a contained POD type field.
- ▶ We can use this information to recover nesting relationships.
  - ▶ Reconstruct not just a flat list of fields, but contained `structs`.

# Dynamic Structure Reconstruction

## Sources of Ambiguity in the Data

Imperfect data (misleading, conflicting, or hard-to-analyze structure field accesses) arises from:

- ▶ Natural causes in the source code.
  - ▶ Casts between integer sizes
  - ▶ Use of `unions`
  - ▶ Arrays
- ▶ Compiler optimizations.
- ▶ Bulk data operations.

A summary of these problems and our solutions follow.

# Dynamic Structure Reconstruction

## Ambiguity #1: Casting

<code>struct X {</code>	<code>int16 b = x-&gt;a;</code>	<code>int8 b = x-&gt;a;</code>
<code>// ...</code>	<code>... might compile to ...</code>	<code>... might compile to ...</code>
<code>int32 a;</code>	<code>mov ax, [rsi+10h]</code>	<code>mov al, [rsi+10h]</code>

- ▶ Casts produce different access sizes to the same field.
- ▶ My solution: choose the most frequently-occurring size.
  - ▶ Works well in this case.

# Dynamic Structure Reconstruction

## Ambiguity #2: Compiler Optimizations

`if(x->flags32 & 0x40)` might compile to:

---

```
F7 06 40 00 00 00 test dword ptr [esi], 40h
```

OR, EQUIVALENTLY:

---

```
F6 06 40 test byte ptr [esi], 40h
```

- ▶ Peephole optimizers may produce the smaller, second one.
- ▶ This can produce different access sizes to the same fields.
- ▶ My solution: same as before. Also works well.

# Dynamic Structure Reconstruction

## Ambiguity #3: Compiler Optimizations and Phantom Fields

`if(x->flags32 & 0x400 ★)` might compile to:

`test dword ptr [esi], 400h ★`

---

OR, EQUIVALENTLY:

`test byte ptr [esi+1], 4h ●`

- ▶ Similar to before, but the **access location** increments by 1.
  - ▶ Notice the different constants `400h ★` vs. `4h ●`.
- ▶ This can produce phantom fields within the structure.
- ▶ My solution: same as before. Also works well.

# Dynamic Structure Reconstruction

## Ambiguity #4: Store Aggregation

```
struct X {  
    // ...  
    bool a;    x->a = 0; ◀    xor ax, ax  
    bool b;    x->b = 0; ◀    mov word ptr [rsi+18h], ax
```

- ▶ Compiler may merge adjacent writes ◀ into a **larger write**.
  - ▶ Produces accesses bigger than the field in question.
- ▶ My solution: choose the most frequently-occurring size.
  - ▶ Works well in this case.

# Dynamic Structure Reconstruction

## Ambiguity #5: Bulk Copies and Assignments

```
struct X {                                memset(x1,0,sizeof(X)) might compile to:
  char a;      ◀                          xor  eax, eax
  char b;      ◀                          mov  qword ptr [rcx], rax ◀
  short c;     ◀                          mov  qword ptr [rcx+8], rax
  int d;       ◀                          mov  qword ptr [rcx+10h], rax
  // ...                                          mov  qword ptr [rcx+18h], rax
                                          ...
```

- ▶ Compiled `memset` and `memcpy` often use block operations, i.e., are oblivious to the sizes/configurations of structure fields.
- ▶ This can produce different access sizes to the same fields.
- ▶ My solution: same as before. Also works well.



# Dynamic Structure Reconstruction

## Ambiguity #6: unions

<pre>struct X {     // ...     union {         int a;         char *b;     }     // ...</pre>	<pre>int c = x-&gt;a; ... might compile to ... mov eax, [rsi+10h]</pre>	<pre>char *d = x-&gt;b; ... might compile to ... mov rax, [rsi+10h]</pre>
---	---	---

- ▶ With `unions`, **multiple variables (of different types and sizes) occupy a single memory location.**
- ▶ Clearly there can be multiple sizes for one location.
- ▶ My solution: as before, choose the most frequent size.
  - ▶ That is, sidestep and ignore `unions` completely.
  - ▶ Not a real solution, and does not work very well!

# Dynamic Structure Reconstruction

## Ambiguity #7: Array Accesses

```
struct X { | int b = x->a[i];  
    // ... |     ... might compile to ...  
    int a[10]; | mov ebx, [rsi+rbx*4+10h]
```

- ▶ Arrays produce references to **non-constant offsets**.
- ▶ My solution: discard accesses with non-constant offsets.
  - ▶ Also not a real solution.
  - ▶ Does not recover arrays, period, let alone do it well.

## Dynamic Structure Reconstruction

Existing DBI-Based Approaches

**Limitations of DBI-Based Solutions**

My Contributions to this Problem

# Limitations of DBI-Based Solutions

## Limitation #1: Overhead from Instrumentation

- ▶ The technique is comprehensive and fully automated, but ...
- ▶ **Every** memory access must be instrumented.
  - ▶ **Every** memory access incurs a map lookup.
  - ▶ This is a relatively heavyweight application of DBI.
  - ▶ The overhead impedes interaction with the application;
  - ▶ Lower interactivity means lower code coverage; and
  - ▶ Limited code coverage means limited applicability.
- ▶ Furthermore, the overhead is fundamental to the approach.
  - ▶ No matter how we optimize it, fundamentally, the approach instruments all memory accesses.

# Limitations of DBI-Based Solutions

## Limitation #2: Overhead from Tracking Every Allocation

- ▶ Every instrumented memory access requires a map lookup.
  - ▶ Binary trees ensure slow  $\log(N)$  growth, but nevertheless ...
  - ▶ ... more allocations tracked means slower lookups.
- ▶ Can we reduce the overhead even further?, and/or
- ▶ Is it useful to track only a subset of allocations?

# Limitations of DBI-Based Solutions

## Friendliness as an Interactive Tool

- ▶ DBI solution is fire-and-forget, monolithic, and slow.
- ▶ Can we make a useful interactive reverse engineering tool?
- ▶ How best to offer the results within Hex-Rays and IDA?
  - ▶ GUIs for browsing the results
  - ▶ Full automation for common type annotation tasks
  - ▶ Library elements for custom tasks

## Directions Explored in this Research

1. Track memory accesses via page-related chicanery.
2. Allow the user to specify particular allocation sites of interest, rather than targeting all at once.
3. Divert interesting allocations into custom allocators.
4. Performance-optimize everything.
5. Make good use of the resulting data.

## Dynamic Structure Reconstruction

Existing DBI-Based Approaches

Limitations of DBI-Based Solutions

My Contributions to this Problem

- Exploit X86 Demand-Based Paging

- DLL Injection-Based Memory Tracking

- Target Specific Allocation Sites

- Exploit the Results within IDA/Hex-Rays

- Target-Specific Example: `unions`



# Focus on the Memory, not the Instructions

## Idea #1: Debug Breakpoints

First idea for replacing DBI: X86 debug/memory breakpoints.

```
mov rcx, 20h
call malloc ; Set 8-byte R/W breakpoint on:
            ; rax+0x00
            ; rax+0x08
            ; rax+0x10
            ; rax+0x18
```

- ▶ PRO: only incurs overhead when the memory is accessed
- ▶ CON: can only set 4 breakpoints, i.e., 0x20 bytes of memory
  - ▶ Would like to track megabytes/gigabytes, not “bytes”.

Verdict: right direction, wrong scale.

# Focus on the Memory, not the Instructions

## Idea #2: Page Breakpoints

Second idea for replacing DBI: SoftICE's BPR feature.

```
mov rcx, 88h
```

```
call malloc
```

SoftICE command:

```
:bpr rax rax+0x88 rw
```

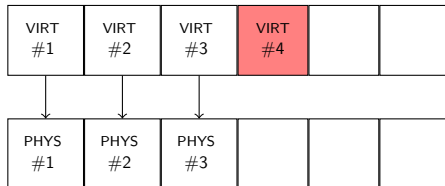
- ▶ Unlimited memory breakpoints of any size!
- ▶ Can be implemented in kernel-mode or user-mode.
- ▶ Note, also implemented by other tools:
  - ▶ IDA's large memory breakpoints
  - ▶ OllyDbg's page breakpoints

Sounds promising! Let's review how it works.

# Virtual Memory and Demand Paging

## Virtual Memory Concept

Page Table Entries (PTEs)

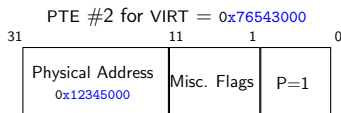
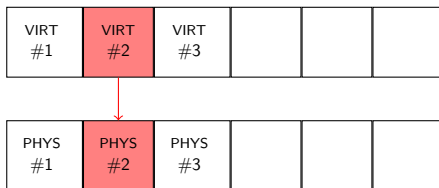


The OS provides the illusion of a large virtual address space, but only mapped addresses are valid. (E.g., #4 is not.)

# Virtual Memory and Demand Paging

## Virtual-to-Physical Address Translation

### Page Table Entries (PTEs)



The page table maintains the virtual-to-physical mappings. E.g. if virtual 0x76543000 is mapped to physical 0x12345000, then

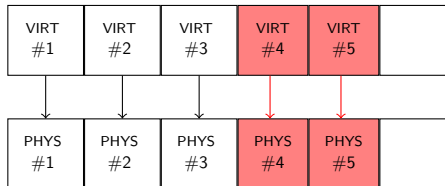
0x76543 012 resolves to 0x12345 012.

VirtualOffset                      PhysicalOffset

# Virtual Memory and Demand Paging

## On-Demand Growth of Virtual Address Space

Page Table Entries (PTEs)



The OS can grow the virtual address space on demand by allocating and mapping additional physical pages.

# Virtual Memory and Demand Paging

## Swapping Under High Memory Load

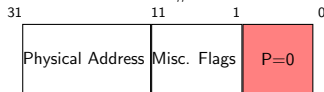
Page Table Entries (PTEs)

VIRT #1	VIRT #2	VIRT #3	VIRT #4	VIRT #5	
---------	---------	---------	---------	---------	--



PHYS #1	PHYS #2	PHYS #3	PHYS #4	PHYS #5	
---------	---------	---------	---------	---------	--

PTE #3



When memory is scarce, the OS reclaims physical pages by:

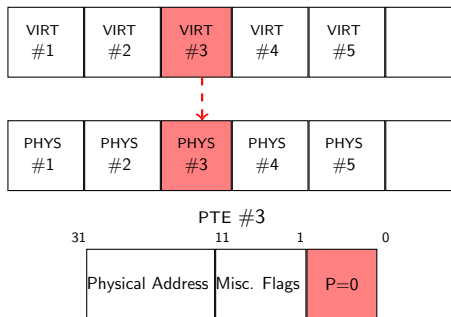
1. Writing their contents to disk.
2. Marking the corresponding PTE entry as non-present (P=0).

Later accesses to non-present pages generate page fault exceptions.

# Virtual Memory and Demand Paging

## Transparently Reloading Swapped Pages

Page Table Entries (PTEs)



In response to page faults in non-present pages, the OS:

1. Allocates a physical page.
2. Loads the page's previous contents from disk.
3. Updates the PTE with the physical address and  $P=1$ .
4. Resumes execution.

## Dynamic Structure Reconstruction

Existing DBI-Based Approaches

Limitations of DBI-Based Solutions

### My Contributions to this Problem

Exploit X86 Demand-Based Paging

DLL Injection-Based Memory Tracking

Target Specific Allocation Sites

Exploit the Results within IDA/Hex-Rays

Target-Specific Example: `unions`



# Memory Tracing via Presence

## Overview

SoftICE's BPR, IDA's large memory, and OllyDbg's page breakpoints co-opt X86's demand paging mechanism as such:

- ▶ Mark pages of interest as non-present.
- ▶ Intercept page fault exceptions for those pages.
- ▶ Determine whether the region of interest was accessed;
- ▶ Break if so, continue execution if not.

We exploit this same mechanism to track structure accesses.

# Memory Tracing via Presence

## Mechanism in Detail

```
61F33E5A mov eax, [ebx+4]
```

```
61F33E60 add edx, 4
```

Before the first instruction executes, assume that the page at `[ebx+4]` has been marked as non-present.

# Memory Tracing via Presence

## Mechanism in Detail

```
▶ 61F33E5A mov eax, [ebx+4]  
61F33E60 add edx, 4
```

First instruction attempts to execute.

# Memory Tracing via Presence

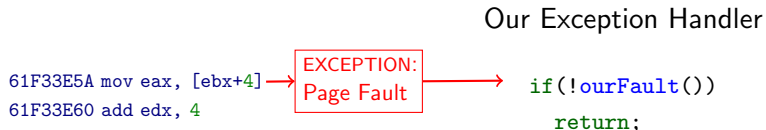
## Mechanism in Detail

```
▶ 61F33E5A mov eax, [ebx+4] → EXCEPTION:  
61F33E60 add edx, 4 Page Fault
```

Since `[ebx+4]` is non-present, the CPU triggers a page fault exception.

# Memory Tracing via Presence

## Mechanism in Detail



The OS transfers control to our exception handler.

# Memory Tracing via Presence

## Mechanism in Detail

```
61F33E5A mov eax, [ebx+4]
61F33E60 add edx, 4
```

EXCEPTION:  
Page Fault

## Our Exception Handler

```
▶ if(!ourFault())
▶ return;
```

If the address was not within a tracked region, we pass.

# Memory Tracing via Presence

## Mechanism in Detail

```
▶ 61F33E5A mov eax, [ebx+4]
  61F33E60 add edx, 4
```

EXCEPTION:  
Page Fault

## Our Exception Handler

```
▶ log(exnDetails);
▶ save(faultEa); ◀
```

Log structure access (identically to what we did for DBI).

Also, record the address of the faulting instruction

(61F33E5A ◀ in this case).

# Memory Tracing via Presence

## Mechanism in Detail

```
61F33E5A mov eax, [ebx+4]  
61F33E60 add edx, 4
```

EXCEPTION:  
Page Fault

## Our Exception Handler

▶ `makePresent(page);`

Mark the page at `[ebx+4]` as present again.



# Memory Tracing via Presence

## Mechanism in Detail

```
61F33E5A mov eax, [ebx+4]  
61F33E60 add edx, 4
```

EXCEPTION:  
Page Fault

## Our Exception Handler

- ▶ `setTrapFlag();`
- ▶ `resumeExecution();`

Set the X86 trap flag (TF). This will allow one instruction to execute, after which a single-step exception will be raised. Continue execution of the monitored program.

# Memory Tracing via Presence

## Mechanism in Detail

```
▶ 61F33E5A mov eax, [ebx+4]  
61F33E60 add edx, 4
```

The first instruction executes again. Since the page at `[ebx+4]` is now present, execution succeeds this time.

# Memory Tracing via Presence

## Mechanism in Detail

```
61F33E5A mov eax, [ebx+4] → EXCEPTION:  
61F33E60 add edx, 4 Single Step
```

The second instruction would execute, but since the TF was previously set, the CPU raises a single-step exception.

# Memory Tracing via Presence

## Mechanism in Detail



The OS invokes our single step exception handler. Ensure that the faulting address ◀ is immediately after the previously-saved faulting address ◀. If not, we pass.

# Memory Tracing via Presence

## Mechanism in Detail

```
61F33E5A mov eax, [ebx+4]
61F33E60 add edx, 4
```

EXCEPTION:  
Single Step

Our Exception Handler

▶ `makeNonPresent(page);`

Mark the page at `[ebx+4]` as non-present again, ensuring that we catch future accesses to the monitored page.

# Memory Tracing via Presence

## Mechanism in Detail

```
61F33E5A mov eax, [ebx+4]
61F33E60 add edx, 4
```

EXCEPTION:  
Single Step

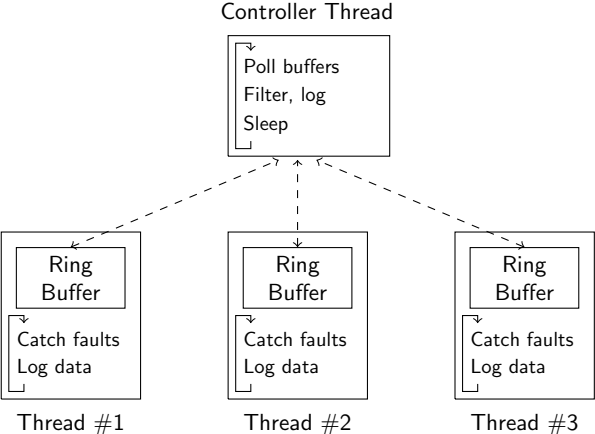
## Our Exception Handler

▶ `resumeExecution();`

Resume execution of the program.

# Memory Tracing via Presence

## Multi-Threaded Architecture



A dedicated thread retrieves events from the program's thread-local data, filters duplicates, and logs the results to disk via buffered I/O.

# Memory Tracing via Presence

## Optimizations

Optimization ideas implemented:

1. Emulate common instructions instead of single-stepping<sup>1</sup>
2. Use guard pages instead of `PAGE_NOACCESS`<sup>2</sup>
3. Force consumer thread away from producer thread cores

Architectural Revision	# Accesses per Minute
Single-Threaded	3M
Multi-Threaded	6.4M
Mini X64 Emulator V1	9M
Guard Pages	11M
<code>SetThreadAffinityMask()</code>	12.1M
Mini X64 Emulator V2	13.2M

---

<sup>1</sup>Suggested by Yaron Dinkin

<sup>2</sup>Suggested by Jason Geffner + RECON attendee whose name I forget (sorry)



## Dynamic Structure Reconstruction

Existing DBI-Based Approaches

Limitations of DBI-Based Solutions

### My Contributions to this Problem

Exploit X86 Demand-Based Paging

DLL Injection-Based Memory Tracking

#### Target Specific Allocation Sites

Exploit the Results within IDA/Hex-Rays

Target-Specific Example: `unions`

# Dynamic Structure Reconstruction

## How to Apply Page-Based Tracking

We've shown how to track memory, but not how to apply it. We explore our two possibilities, and strategies for those cases:

1. Track every allocation, a la DBI.
2. Only track certain allocations.

# Dynamic Structure Reconstruction

## Tracking Every Allocation

When tracking all allocations:

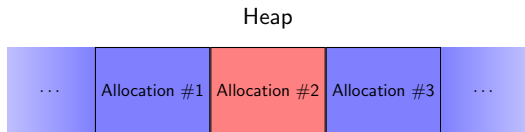
```
void *allocHook(int size) {  
    // existing hook code  
    addBpt(mem, size); ◀  
    // existing hook code  
}  
  
void freeHook(void *mem) {  
    // existing hook code  
    delBpt(mem, size); ◀  
    // existing hook code  
}
```

Simply add breakpoints upon `malloc`, and remove upon `free`.

# Target Specific Allocation Sites

## Noisiness of Technique

When tracking only some allocations:

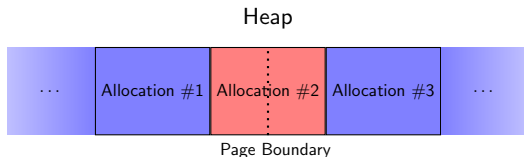


- ▶ Since our technique works at the level of pages, we incur page faults for any allocation on the same page.
  - ▶ Only interested in **red** region.
  - ▶ However, we take faults in **blue** regions on the same page.
  - ▶ More page faults = more overhead.

# Target Specific Allocation Sites

## Noisiness of Technique

When tracking only some allocations:



- ▶ Since our technique works at the level of pages, we incur page faults for any allocation on the same page.
  - ▶ Only interested in **red** region.
  - ▶ However, we take faults in **blue** regions on the same page.
  - ▶ More page faults = more overhead.
- ▶ Worse, we may monitor multiple pages per allocation.
- ▶ Best performance: **only** fault on interesting allocations.

# Target Specific Allocation Sites

## Divert into Custom Allocator

When tracking only specific allocations, to improve performance:

```
void *mallocHook(int size) {  
    if(isInteresting(_RetAddr())) { ◀  
        mem = customAlloc(size); ◀  
        addBpt(mem,size); ◀  
    } else  
        mem = originalMalloc(size);  
    return mem;  
}  
  
void freeHook(void *mem) {  
    if(isCustom(mem)) { ◀  
        delBpt(mem,size); ◀  
        customFree(mem); ◀  
    } else  
        originalFree(size);  
}
```

Divert interesting allocations into a custom allocator.

Benefit: no page faults for uninteresting allocations!

# Target Specific Allocation Sites

Divert into General-Purpose, Off-the-Shelf Allocator

One strategy for custom allocation: use an existing allocator.

```
HANDLE hHeap = HeapCreate(...);  
//...  
void *customAlloc(size) {  
    return HeapAlloc(hHeap,0,size);  
}
```

Pros:

- ▶ Easy to implement
- ▶ Usually thread-safe
- ▶ Naturally handles different-sized allocations
- ▶ Tuned for performance

```
void customFree(void *mem) {  
    HeapFree(hHeap,0,mem);  
}
```

Cons:

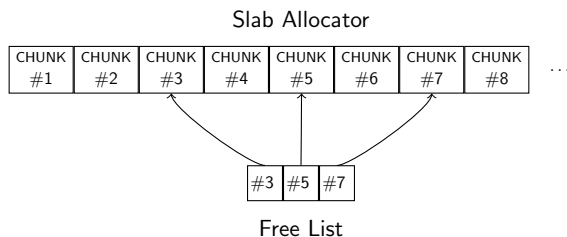
- ▶ Page faults for in-band metadata
- ▶ Slower than some alternatives

# Target Specific Allocation Sites

## Divert into Customized Slab Allocator

Another allocation strategy: fixed-size slab allocator.

---



### Pros:

- ▶ Fast allocation and range checks
- ▶ No in-band metadata

### Cons:

- ▶ Fixed-size
- ▶ Must be applied judiciously



# Dynamic Structure Reconstruction

Summary: DBI vs. DLL Injection

## DBI

1. Hook `malloc/free`
2. Record allocation details
3. Instrument memory references
4. Keep accesses to allocations
5. Log structure accesses
6. Post-process data

## DLL Injection

1. Hook `malloc/free`
2. Divert chosen allocations into custom allocator
3. Mark allocations non-present
4. Catch memory exceptions
5. Log structure accesses
6. Post-process data

## Dynamic Structure Reconstruction

Existing DBI-Based Approaches

Limitations of DBI-Based Solutions

### My Contributions to this Problem

Exploit X86 Demand-Based Paging

DLL Injection-Based Memory Tracking

Target Specific Allocation Sites

Exploit the Results within IDA/Hex-Rays

Target-Specific Example: `unions`

# Dynamic Structure Reconstruction

## Loading the Data in IDA

Load raw structure accesses (unknown type)

Load raw structure accesses (known type)

Load raw union accesses

Import function argument data

Import allocation site types

Export allocation site types

Font...

Right-click in the main GUI window.

- ▶ **Unknown structure:** for structure recovery purposes.
- ▶ **Known structure:** when the structure type is already known.
  - ▶ This case is still very useful, as we will see.

# Dynamic Structure Reconstruction

## List of Raw Structure Accesses

Function	Address	Raw	Base	Offset	Disassembly
sub_170BC450	0x170bc452	0x18	0x0	0x18	mov [rcx+18h], edx
sub_170BC450	0x170bc455	0x8	0x0	0x8	mov [rcx+8], rax
sub_170BC450	0x170bc459	0x10	0x0	0x10	mov [rcx+10h], rax
sub_170BC450	0x170bc45d	0x0	0x0	0x0	mov [rcx], rax
sub_1706CF00	0x1706cf0f	0x0	0x0	0x0	mov [rdx], r8d
sub_1706CF00	0x1706d000	0x18	0x0	0x18	mov r9d, [rbx+18h]
sub_1706CF00	0x1706d00b	0x0	0x0	0x0	mov dword ptr [rbx], 4
sub_17149E60	0x17149e8d	0x20	0x20	0x0	cmp byte ptr [rbx], 2
sub_17144EB0	0x17144ec7	0x20	0x20	0x0	movzx eax, byte ptr [rcx]

Line 20 of 6070

After loading the access data for some allocation site(s).

# Dynamic Structure Reconstruction

## Raw Structure Access Operations

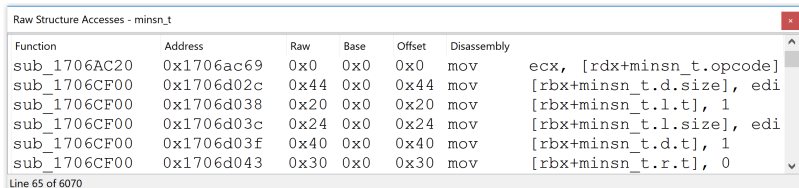
Apply structure references to disassembly (all)  
Apply structure references to disassembly (selected)  
Apply types to Hex-Rays variables (all)  
Apply types to Hex-Rays variables (selected)  
Font...

Right-clicking provides two primary operations:

1. Apply structure offsets in the disassembly.
2. Change the types of variables in Hex-Rays.

# Dynamic Structure Reconstruction

## Applying Structure Offsets in the Disassembly



Function	Address	Raw	Base	Offset	Disassembly
sub_1706AC20	0x1706ac69	0x0	0x0	0x0	mov ecx, [rdx+minsn_t.opcode]
sub_1706CF00	0x1706d02c	0x44	0x0	0x44	mov [rbx+minsn_t.d.size], edi
sub_1706CF00	0x1706d038	0x20	0x0	0x20	mov [rbx+minsn_t.l.t], 1
sub_1706CF00	0x1706d03c	0x24	0x0	0x24	mov [rbx+minsn_t.l.size], edi
sub_1706CF00	0x1706d03f	0x40	0x0	0x40	mov [rbx+minsn_t.d.t], 1
sub_1706CF00	0x1706d043	0x30	0x0	0x30	mov [rbx+minsn_t.r.t], 0

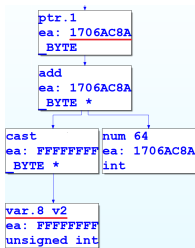
Line 65 of 6070

Applying structure offsets is extremely fast.



# Dynamic Structure Reconstruction

## Locating Hex-Rays Variables



1706AC8A movzx eax, byte ptr [rbx+40h]

For a given discovered structure access, locate the **pointer dereference at that address** in the Hex-Rays CTREE.

Does not always work. Hex-Rays may:

- ▶ Lose track of the address of the memory dereference;
- ▶ Not create a variable for the pointer dereference;
- ▶ Render the access via many patterns, some of which I miss.

Can be improved somewhat, but will never be perfect.



# Dynamic Structure Reconstruction

## Applying Hex-Rays Variable Types

```
v4 = (_DWORD *)qalloc_or_throw(80i64);
v5 = (__int64)v4;
if ( v4 )
{
    v4[offsetof(minsn_t, next)] = 0;
    v4[9] = -1;
    v4[0xC] = 0;
    v4[0xD] = -1;
    v4[offsetof(minsn_t, prev)] = 0;
    v4[0x11] = -1;
    sub_170BC450((__int64)v4, -1);
}
```

```
v4 = (minsn_t *)qalloc_or_throw(80i64);
v5 = v4;
if ( v4 )
{
    v4->l.t = 0;
    v4->l.oprops = 0;
    v4->l.valnum = 0;
    v4->l.size = -1;
    v4->r.t = 0;
    v4->r.oprops = 0;
    v4->r.valnum = 0;
    v4->r.size = -1;
    v4->d.t = 0;
    v4->d.oprops = 0;
    v4->d.valnum = 0;
    v4->d.size = -1;
    sub_170BC450((__int64)v4, -1);
}
```

Comparison before and after applying Hex-Rays types.

## Dynamic Structure Reconstruction

Existing DBI-Based Approaches

Limitations of DBI-Based Solutions

### My Contributions to this Problem

Exploit X86 Demand-Based Paging

DLL Injection-Based Memory Tracking

Target Specific Allocation Sites

Exploit the Results within IDA/Hex-Rays

Target-Specific Example: **unions**

# Dynamic Structure Reconstruction

## unions

```
union U {  
    int x;  
    char *y;  
    void *z;  
};
```

- ▶ unions allow multiple interpretations of the same variable.
- ▶ U can hold **either** an `int`, `x`, **OR** a `char *`, `y`, **OR** a `void *`, `z`.

# Dynamic Structure Reconstruction

## Tagged unions

```
enum Uelt {
    Uint = 0,
    Ucptr = 1,
    Uvptr = 2
};

struct taggedU {
    Uelt t;
    U e;
};
```

- ▶ The **tagged union** pattern associates an `enum` with a `union`.
  - ▶ `t` is called the **tag** or the **discriminant**.
- ▶ This design pattern is especially common in programming language tools (compilers, interpreters, decompilers, etc).

# Dynamic Structure Reconstruction

## Tag Checking

```
void print(taggedU *u) {  
    if(u->t == Uint)   printf("%d\n", u->e.x);  
    if(u->t == Ucptr)  printf("%s\n", u->e.y);  
    if(u->t == Uvptr)  printf("%llx\n", u->e.z);  
};
```

- ▶ The code must check the tag to know the `union`'s held type.
  - ▶ Code using `unions` is **littered** with these checks.

# Dynamic Structure Reconstruction

## unions in Decompilation: Improper Selection

```
v162 = (__int64)&v8->l;  
v163 = v8->l.nnn;  
if ( v163->ea - 8 <= 1 )  
    v162 = (__int64)&v163[1].value;  
if ( *(__BYTE *)v162 == 4  
    && **(__DWORD **)(v162 + offsetof(mop_t, u4)) == 10  
    && (v154 == (mop_t **)255 || v154 == (mop_t **)0xFFFF  
{  
    v164 = (int)sub_61F66290((unsigned int)v154) / 8;  
    if ( *(__DWORD *) (v162 + 4) >= v164 )  
    {  
        *(__DWORD *) (v162 + 4) = v164;  
        *(__DWORD *) (*(__QWORD *) (v162 + 8) + 68i64) = v164;
```

Proper `union` field selection is critical to readable decompilation.  
Failure to do so leads to hideous code like this.

# Dynamic Structure Reconstruction

## unions in Decompilation: Proper Selection

```
v163 = &v8->l;  
v164 = v8->l.d;  
if ( (unsigned int) (v164->opcode - 8) <= 1 )  
    v163 = &v164->l;  
if ( v163->op == 4  
    && v163->d->opcode == 10  
    && (v155 == (mop_t **)255 || v155 == (mop_t **)0xFFFF ||  
{  
    v165 = (signed int)sub_61F66290((unsigned int)v155) / 8;  
    if ( v163->size >= v165 )  
    {  
        v163->size = v165;  
        v163->d->d.size = v165;
```

Same code as the previous, with three `union` fields set properly.  
`unions` are particularly tedious to apply manually – let's automate.

# Dynamic Structure Reconstruction

Upon Manually Discovering a `union` Somewhere ...

```
class mop_t {  
+0x00     mopt_t t; ◀  
+0x01     char oprops;  
+0x02     short valnum;  
+0x04     int size;  
+0x08     union { ... }; ◀  
+0x10 };
```

- ▶ Suppose we discover a tagged `union` within an allocated type.
  - ▶ Suppose we discover the tag location at `+0x00` ◀.
  - ▶ Suppose we discover the `union` location at `+0x08` ◀.
- ▶ Run the structure access discovery again, but this time:
  - ▶ Only log accesses to the `union` region. ◀
  - ▶ Log the value of the `enum` at every `union` access. ◀



# Dynamic Structure Reconstruction

Determine the Number of Union Variants

RVA	enum	Type
150F03	0x2	READ
151F8A	0xA	READ
1520BC	0xC	READ
14E9AE	0x8	READ
15EB78	0x4	READ
1531A8	0x4	READ
146F01	0xE	READ

This data helps determining the number of union variants. Create a `union` with this many variants (of the observed sizes).

# Dynamic Structure Reconstruction

Mapping between `enum` Elements and `union` Fields

```
enum mop_t {
    mop_z = 0,
    mop_r = 1,
    mop_n = 2,
    mop_str = 3,
    mop_d = 4,
    mop_S = 5,
    mop_v = 6,
    mop_b = 7,
    mop_f = 8,
    mop_l = 9,
    mop_a = 10,
    mop_h = 11,
    mop_c = 12,
    mop_fn = 13,
    mop_p = 14,
    mop_sc = 15
};

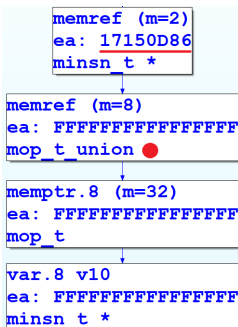
union {
    mreg_t r;
    mnumber_t *nnn;
    mins_t *d;
    stkvar_ref_t *s;
    ea_t g;
    int b;
    mfuncinfo_t *f;
    lvar_ref_t *l;
    mop_addr_t *a;
    char *helper;
    char *cstr;
    mcases_t *c;
    fnumber_t *fpc;
    mop_pair_t *pair;
    scif_t *scif;
};
```

Through reverse engineering, manually establish a mapping between the `enum` elements and `union` variant numbers.

# Dynamic Structure Reconstruction

## Setting Hex-Rays Union Selections

```
17150D86 mov rbx, [rbx+28h]
```



1. First, set the type of the base variable, as before.
2. Next, locate the union reference ●.
3. Finally, apply the proper union element number, based on the tag value recorded at runtime.

## Dynamic Resolution of Argument Types

Preprocessing

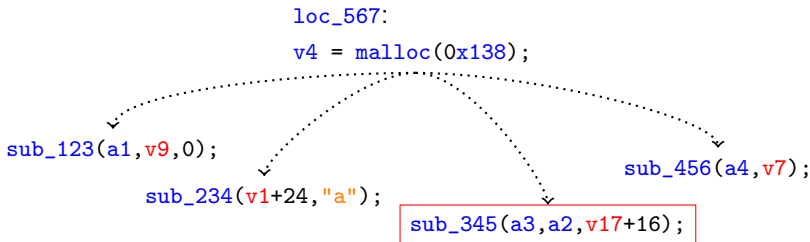
Run-Time Data Collection

Applying the Results

# Dynamic Resolution of Argument Types

## Overview: Big Picture

Via DLL injection, for every call to `malloc`, record the pointer.



Discover every function receiving an allocated pointer as argument.  
Record  $\langle$  function RVA, arg #, allocation RVA, size, pointer offset  $\rangle$ .

E.g., record  $\langle$  0x345, #3, 0x567, 0x138, 16  $\rangle$ .

# Dynamic Resolution of Argument Types

## Preprocessing #1: Locate Functions via x64 Exception Directory

```
RUNTIME_FUNCTION <rva sub_61EB80C0 ◀, rva algn_61EB80DB ◀, rva stru_620C0390>
```

```
61EB80C0 sub_61EB80C0 ◀proc near  
61EB80C0 var_18= qword ptr -18h  
61EB80C0 sub rsp, 38h  
61EB80C4 mov [rsp+38h+var_18], -2  
61EB80CD mov rcx, [rcx+60h]  
61EB80D1 call free  
61EB80D6 add rsp, 38h  
61EB80DA retn  
61EB80DA sub_61EB80C0 endp  
61EB80DB algn_61EB80DB: align 20h ◀
```

- ▶ The PE64 Exception Directory has `RUNTIME_FUNCTION` entries.
  - ▶ These give the beginning of every non-leaf function ◀,
  - ▶ and its end (or the beginning of its first `try` block) ◀.

# Dynamic Resolution of Argument Types

## Preprocessing #2: Filter Unusable Functions

```
▶ 61EB80C0    sub rsp, 38h
   61EB80C4    mov [rsp+38h+var_18], -2
   61EB80CD    mov rcx, [rcx+60h]
```

Iterate through the function's instructions. **FAIL** if instruction:

- |   |  |   |
|---|--|---|
| t | <ul style="list-style-type: none"><li>▶ Cannot be decoded</li><li>▶ Has control-flow</li><li>▶ Is not easily relocatable</li></ul> | <ul style="list-style-type: none"><li>▶ Has incoming cross-references</li><li>▶ Is after function end / beginning of next <b>try</b> block (per X64 exception metadata)</li></ul> |
|---|--|---|

# Dynamic Resolution of Argument Types

## Preprocessing #2: Filter Unusable Functions

```
61EB80C0    sub rsp, 38h
▶ 61EB80C4    mov [rsp+38h+var_18], -2
61EB80CD    mov rcx, [rcx+60h]
```

Iterate through the function's instructions. **FAIL** if instruction:

- |   |  |   |
|---|--|---|
| t | <ul style="list-style-type: none"><li>▶ Cannot be decoded</li><li>▶ Has control-flow</li><li>▶ Is not easily relocatable</li></ul> | <ul style="list-style-type: none"><li>▶ Has incoming cross-references</li><li>▶ Is after function end / beginning of next <b>try</b> block (per X64 exception metadata)</li></ul> |
|---|--|---|



# Dynamic Resolution of Argument Types

## Preprocessing #2: Filter Unusable Functions

```
61EB80C0    sub rsp, 38h
61EB80C4    mov [rsp+38h+var_18], -2
▶ 61EB80CD    mov rcx, [rcx+60h]
```

Iterate through the function's instructions. **FAIL** if instruction:

- |                             |   |
|-----------------------------|---|
| ▶ Cannot be decoded         | ▶ Has incoming cross-references   |
| ▶ Has control-flow          | ▶ Is after function end / beginning of next <code>try</code> block (per X64 exception metadata) |
| ▶ Is not easily relocatable |   |

Succeed after `sizeof(call)` bytes. ◀

# Dynamic Resolution of Argument Types

## Preprocessing #3: Force `__fastcall` Calling Convention

For each kept function, get the prototype from Hex-Rays.

<code>signed __int64</code>				<code>signed __int64</code>	
<code>__usercall@&lt;rax&gt;</code>		●		<code>__fastcall</code>	●
<code>sub_17078400(</code>				<code>sub_17078400(</code>	
<code>  @__int64 a1&lt;rdx&gt;,</code>		◀		<code>  __int64 a2,</code>	▶
<code>  @__int64 a2&lt;rcx&gt;,</code>		▶		<code>  __int64 a1)</code>	◀
<code>  signed int a3&lt;r15d&gt;)</code>		◀			

Remove non-`__fastcall`-compliant arguments; reorder remaining.

# Dynamic Resolution of Argument Types

## Preprocessing #4: Record Positions of Pointer-Sized Arguments

```
void __fastcall
sub_61EB8AD0(
    void *rcx0, ◀
    unsigned int a2,
    void *a3) ◀
```

For each pointer-sized argument ◀, record positions (#0, #2).  
(Standardized X64 `__fastcall` makes this easier than on X86.)  
Discard functions with no pointer-sized arguments.

Determining argument sizes isn't perfect; Hex-Rays sometimes makes mistakes.

# Dynamic Resolution of Argument Types

## Preprocessing Summary

Function RVA	# Prolog Bytes	# Args	# arg <sub>0</sub>	# arg <sub>1</sub>	...
--------------	----------------	--------	--------------------	--------------------	-----

For each suitable function with pointer-size arguments, record:

- ▶ Function's location
- ▶ Number of prolog bytes
- ▶ Number, positions of pointer-sized arguments

---

Function RVA	# Prolog Bytes	# Tracked Args	Arg positions			
0xe760	5	4	0	1	3	4
0x114e50	8	3	0	1	2	
0xf6f60	5	3	0	1	3	
0x47c20	6	2	0	1		
0x10c4d0	5	2	0	1		

## Dynamic Resolution of Argument Types

Preprocessing

**Run-Time Data Collection**

Applying the Results

# Dynamic Resolution of Argument Types

## Step #1: Hook Memory Management Functions

Hook allocators via DLL injection.

```
.idata:61EB19C8 extrn __imp_free:dword HOOK → &freeHook  
.idata:61EB19CC extrn __imp_malloc:dword → &mallocHook  
HOOK
```

As before, record allocation records from `malloc` until `free`.

Allocation Record

Allocated pointer	Size	RVA of return address from <code>malloc</code>
-------------------	------	--

# Dynamic Resolution of Argument Types

## Step #2: Hook Every Suitable Function

```
▶ .text:61F6ABD0  mov  r8, rcx
▶ .text:61F6ABD3  push rbx
▶ .text:61F6ABD4  sub  rsp, 80h
▶ .text:61F6ABDB  mov  [r11-68h], -2

▶ .text:6205CC10  push rdi
▶ .text:6205CC12  sub  rsp, 40h
▶ .text:6205CC16  mov  [rsp+48h+var_28], -2

▶ .text:6205F91C  sub  rsp, 18h
▶ .text:6205F920  mov  r8, rcx
▶ .text:6205F923  mov  eax, 5A4Dh

▶ .text:61F79570  mov  rax, rsp
▶ .text:61F79573  push r14
▶ .text:61F79575  sub  rsp, 60h

▶ .text:61F00AA0  sub  rsp, 38h
▶ .text:61F00AA4  mov  [rsp+38h+var_18], -2
▶ .text:61F00AAD  mov  rcx, [rcx]
```

For each function to hook ...

# Dynamic Resolution of Argument Types

## Step #2: Hook Every Suitable Function



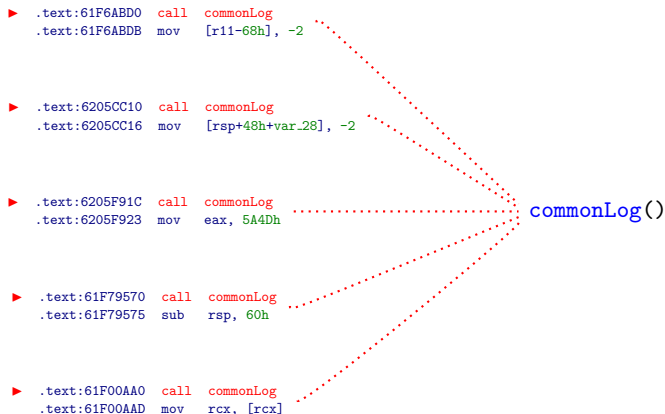
Allocate memory for re-entry thunks. Copy the leading instructions, and insert a jump to after the copied instructions.

Record { Original RVA, Thunk VA } in a hash table.



# Dynamic Resolution of Argument Types

## Step #2: Hook Every Suitable Function



Divert every function into a **common logging routine**.

# Dynamic Resolution of Argument Types

## Redirect Functions into a Common Logging Stub

```
commonLog:
; Save flags
; Save registers
mov rcx, rsp          ; Point arg #0 to stack data
call commonLogC
mov [rsp+78h], rax ◀ ; Return to re-entry thunk
; Restore registers
; Restore flags
retn
```

- ▶ `commonLog` just invokes its C counterpart.
  - ▶ Overwrite return address ◀ with function's re-entry thunk.

# Dynamic Resolution of Argument Types

## Argument Logging Details

```
uint64_t commonLogC(uint64_t *args) {  
    funcRVA = _ReturnAddress();  
#1    reEntry, argList = lookup(funcRVA);  
#2    for(argNo : argList)  
#2        log(funcRVA, argNo, args[argNo]);  
#3    return reEntry;  
}
```

1. Fetch re-entry address, list of interesting arguments.
2. Log each interesting argument.
  - ▶ This happens in another thread for efficiency.
3. Return to re-entry thunk.

# Dynamic Resolution of Argument Types

Filter, Log Allocation Flow to Arguments

```
void log(uint64_t funcRVA, int argNo, uint64_t arg) {  
#1   allocRec = allocMapLookup(arg);  
#2   if(!allocRec) return;  
#3   write(funcRVA, argNo, allocRec);  
}
```

1. Look up function argument in allocation map.
2. Return if not part of an allocation.
3. Write log entry otherwise.

## Logged Data

Function RVA	# Arg	Alloc RVA	Alloc size	Offset into alloc
--------------	-------	-----------	------------	-------------------

# Dynamic Resolution of Argument Types

Summary: Logged Data

Logged Data

Function RVA	# Arg	Alloc RVA	Alloc size	Offset into alloc
0x15f520	1	0xbe648	0x50	0x20
0x153ce0	0	0xbe648	0x50	0x0
0x147520	0	0x143fd8	0x50	0x40
0x15f8d0	1	0x143fd8	0x50	0x40
0x11530	0	0x56c89	0x630	0x0
0x55a80	0	0x56149	0x120	0x0
0x57bd0	0	0x56149	0x120	0x18

This generates a **lot** of data (~60K entries for my target).

## Dynamic Resolution of Argument Types

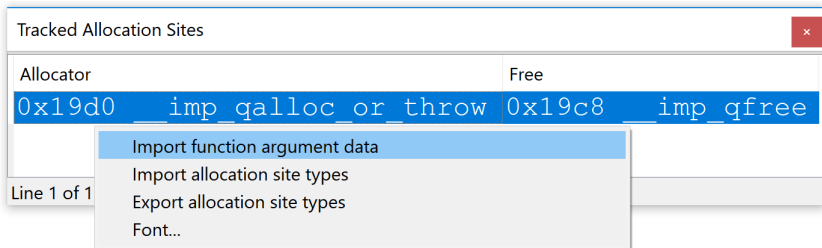
Preprocessing

Run-Time Data Collection

Applying the Results

# Dynamic Resolution of Argument Types

## Loading the Data in IDA



- ▶ First, create an allocator/free pair.
  - ▶ Can add multiple allocators if appropriate.
- ▶ Next, load the data for that allocator.

# Dynamic Resolution of Argument Types

## Displaying Allocation Sites and Types

Allocation Site	Size	Varia	Type
0x1707a5fc	0x30	N	
0x170ad600	0x8	Y	
0x17150002	0x10	N	
0x170e9803	0x10	N	
0x17048e04	0x70	N	
0x1706f006	0x50	N	
0x1714f407	0x50	N	

Line 1 of 270

- ▶ Double-click an allocator to see the list of data:
  1. The address of all observed allocation sites
  2. The size allocated by that site
    - ▶ If multiple sizes, show the GCD
  3. The user-supplied type for that site, if any



# Dynamic Resolution of Argument Types

## Displaying Allocation Site Flow Data

Function	Arg #	Offset	Size
sub_170735A0	1	0x0	0x30
sub_17165AA0	1	0x0	0x30
sub_17048840	0	0x0	0x30
sub_1704D5F0	3	0x0	0x30
sub_170DCC30	1	0x0	0x30
sub_17111040	0	0x0	0x30
sub_1713BCD0	1	0x0	0x30

Line 1 of 155

- ▶ Double-click an allocation site to see the list of:
  1. Functions and arguments into which the allocations flowed
  2. Observed offsets from the base of the allocation

# Dynamic Resolution of Argument Types

## Displaying Allocation Flow in Hex-Rays

Pseudocode-A

```
1// Dynamic allocation flow data
2// 0
3//     0x1717b823[0x138]
4//     0x17181421[0x138]
5//     0x17180fb1[0x138]
6// 1
7//     0x1717b823[0x138]+216
8//     0x17180fb1[0x138]+88
9//     0x17181421[0x138]+88
10//    0x1717b823[0x138]+88
11//    0x1717b823[0x138]+152
12//    0x17181421[0x138]+152
13//    0x17181421[0x138]+216
14//    0x17180fb1[0x138]+216
15//    0x17180fb1[0x138]+152
16 __int64 __fastcall sub_171173C0(__int64 a1, __int64 a2, int a3)
```

Hex-Rays listings will automatically display allocation flow data.

# Dynamic Resolution of Argument Types

## Setting Allocation Site Types

```
v10 = ( _DWORD *)galloc or throw(80i64);  
if ( v10 )
```

Set call type...

Once known, the user manually sets the allocation site type.

(Or, in IDA: Edit->Operand Type->Set Operand Type)

---

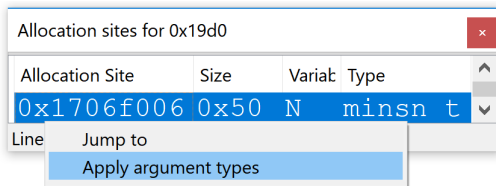
Allocation Site	Size	Variation	Type
0x1706f006	0x50	N	minsn_t

Line 2 of 270

After refresh, the allocation sites window shows the type.

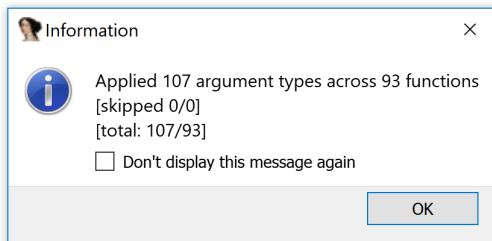
# Dynamic Resolution of Argument Types

## Applying Argument Types



For known allocation site types, the user can apply argument types.

Can select multiple allocation sites at once.



# Dynamic Resolution of Argument Types

## Applied Argument Types

```
char __fastcall sub_17193950(__int64 a1, mins_t *a2, mop_t *a3, mop_t *a4, mop_t *a5)
```

Way better than doing it by hand, isn't it?

# Dynamic Resolution of Argument Types

## Related Types

Offset	Type Name	Func EA	Arg #	Alloc RVA	Alloc Offset	Size
0x0	mblock_t	0x17141040	0	0x170bcac4	0x0	0x178
0x98	bitset_t	0x1700cf00	0	0x170bcac4	0x98	0x178
0x70	bitset_t	0x170112e0	0	0x170bcac4	0x70	0x178
0x70	bitset_t	0x170112e0	0	0x170bcac4	0xc0	0x178
0xc0	bitset_t	0x170112e0	0	0x170bcac4	0x70	0x178
0xc0	bitset_t	0x170112e0	0	0x170bcac4	0xc0	0x178
0xe8	unsigned __int64 *	0x170112e0	1	0x170bcac4	0xe8	0x178

Line 235 of 381

For a given allocation site, for each offset passed to a function argument, display the types of other structure fields passed to the same argument.

## Further Extensions and Challenges

Extensions

Challenges

# Further Extensions

## Combination with Static Analysis

Access data only covers observed behaviors.  
E.g., will not discover the **accesses** ◀ below.

```
// ALWAYS OBSERVED TAKEN  
if(v1 != 0) {  
    x->f0 = 1234;  
    x->fC = 0;  
} // NEVER OBSERVED TAKEN  
else {  
    x->f4 = 456; ◀  
    x->f8 = 789; ◀  
}
```

x

Rename lvar...	N
Set lvar type...	Y
Convert to struct *...	
Create new struct type...	

Can use Hex-Rays **struct** analysis to discover other **accesses** ◀.



# Further Extensions

## Writes-Pointer-To Tracking

When writing a pointer-sized value into a tracked allocation:

```
.text:170B3AD6 mov [rax+8], rdx ◀
```

If `rdx` points within a known allocation, log the details.

This can help determining the pointer types of structure fields.

# Further Extensions

## Maximum Size of Pointed-To Objects

Want to know: how big is the thing an argument points to?

Offset	Size	Max
0x00	0x30	0x30
0x40	0x50	0x10
0x2C	0x30	0x04

Example data reaching a function argument

- ▶ Maximum size is the distance from the offset to the end.
- ▶ Take the **minimum** across all data points.

# Further Extensions

## Inheritance Discovery by Access Location

Derived classes must construct base classes first:

```
GraceWireGeneric *__thiscall GraceWireGeneric__Constructor(  
{  
    GraceObject__Constructor(&this->vtbl, GraceObjectVariety_Wire);
```

```
0042CFF0  GraceWireGeneric__Constructor proc near
```

```
...
```

```
0042D020  push 4
```

```
0042D025  call GraceObject__Constructor
```

# Further Extensions

## Inheritance Discovery by Access Location

```
GraceObject *__thiscall GraceObject__Constructor(  
{  
    this->iVarietyEnum = aVariety;  
    this->vtbl = &GraceObject::`vftable';
```

```
00424096 mov [edi+8], eax  
00424099 lea eax, [edi+0Ch]  
0042409C push eax  
0042409D mov [ebp+a5], eax  
004240A0 mov dword ptr [edi], offset vtbl
```

- ▶ Hence, every class in a single-inheritance hierarchy should have the **same address for its first access**.
- ▶ **CAVEAT**: inlined constructors will break this.

## Further Extensions and Challenges

Extensions

Challenges

# Challenges

## Code Coverage

**As with any dynamic analysis**, results limited to covered code.

```
if(v1 != 0) { // ALWAYS OBSERVED TAKEN
    x->f0 = 1234;
    x->fC = 0;
} else { // NEVER OBSERVED TAKEN
    neverExecutedFunc1(x); ◀
    neverExecutedFunc2(x); ◀
}
```

I offer no real contributions here, other than that the performance optimizations hereinbefore can increase observations per time unit.

# Challenges

## Type-Related Ambiguity

Suppose multiple allocation sites/sizes flow to an argument.

```
class x {  
    int a; ◀  
    int b; ◀  
};
```

+0	int a; ◀
+4	int b; ◀
+8	int c; ◀
+12	int d; ◀

```
class y :  
    public x {  
    int c; ◀  
    int d; ◀  
};
```

What “type” should we assign the argument?

Need inheritance/composition relationships to fully resolve.

The data is still useful without knowing that, though.

# Challenges

## Type- and Size-Related Ambiguity

Suppose multiple allocation sites/sizes flow to an argument.

```
class x {  
    int a;  
    int b;  
};
```

```
class y :  
    public x {  
    char *c;  
};
```

```
class z :  
    public x {  
    void *d;  
};
```

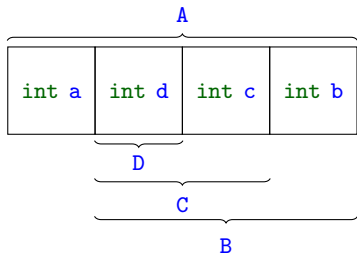
For derived objects, same size does not imply same type.



# Challenges

## Nested Structures

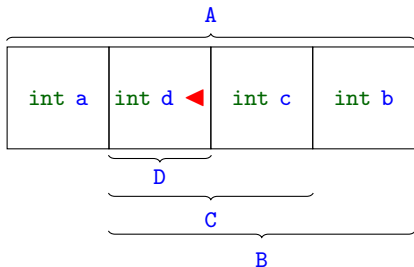
```
struct A {  
    int a;  
    struct B {  
        struct C {  
            struct D {  
                int d;  
            } D;  
            int c;  
        } C;  
        int b;  
    } B;  
};
```



An example of nested structures.

# Challenges

## Access to Nested Structure Fields



Type	Expression
<code>int *</code>	<code>*x</code>
<code>D *</code>	<code>x-&gt;d</code>
<code>C *</code>	<code>x-&gt;D.d</code>
<code>B *</code>	<code>x-&gt;C.D.d</code>

- ▶ Suppose `x` points to `int d` within a `struct A`.
- ▶ What is the C-level type of `x` and the accessing expression?
  - ▶ The four possibilities are shown at right.
- ▶ Even if we had the structure types and nesting relationships from the source code, how would we know the type of `x`?

Introduction

Dynamic Structure Reconstruction

Dynamic Resolution of Argument Types

Further Extensions and Challenges

Conclusion

## Conclusion

- ▶ None of these techniques are particularly sophisticated.
- ▶ However, they are easy-to-use and produce very useful results.
  - ▶ Despite challenges and open problems, the results are useful.
  - ▶ Automation was a better use of my RE time than reading code.
- ▶ Code needs cleanup, but will be released soon.
  - ▶ Check Twitter, Reverse Engineering reddit, etc.

Any Questions?