nccgroup

NCC Group Whitepaper

Automated Reverse Engineering of Relationships Between Data Structures in C++ Binaries

December 7, 2017 – Version 1.0

Prepared by
Nick Collisson

Abstract

Real-time, memory-level interoperability with a closed-source binary may be desired for a number of reasons. In order to read from and write to specific data structures within a target process' memory, external software must have knowledge of how to access these structures at any given time. Since many objects are allocated randomly on the heap, efficiently locating a given piece of data requires the traversal of data structures via a sequence of pointers and offsets that lead from a predictable address to the data of interest (i.e. in the same manner the target application accesses its own data structures). This paper discusses a general approach for finding these kinds of pointer sequences and introduces a new tool implementing this approach.

Table of Contents

1 Introduction . . . 3

2 Prior Art . . . 6

2.1 Techniques . . . 6

2.2 Assessment . . . 6

3 Pointer Sequence Reverser . . . 7

3.1 Design and Implementation . . . 7

3.2 The Debugger . . . 7

3.3 The Tracer . . . 8

3.4 Trace Minimization . . . 10

3.5 Value Column . . . 10

3.6 Vtable Identification . . . 11

3.7 Support for x86, x64, and Position-independent Code . . . 11

4 Case Study . . . 13

4.1 Locating Target Data . . . 13

4.2 Obtaining a Suitable Address . . . 14

4.3 Analyzing PSR Output and Identifying Structures . . . 15

5 The End . . . 20

5.1 Conclusion . . . 20

5.2 Acknowledgements . . . 20

1 Introduction

This paper introduces Pointer Sequence Reverser (PSR), a new tool that shows how an application accesses specific data in its memory. The data can be an object, a variable, or some other data structure. Knowing how an application accesses its data can be useful as this allows one to interoperate with a target binary at runtime, or write tools to do so.

This can be useful when reverse engineering a binary to identify relevant code accessing specific data, to track the locations of other instances of interesting data, or to build tooling that modifies data or functionality within a process (e.g. with function hooking).

Assuming one has sufficient process access privileges, an initial hurdle to the tasks mentioned above is identifying the memory location of a piece of observed data. One method of obtaining the address of a given piece of known data is through memory scanning. Memory scanning tools obtain access to a target application's memory space and allow a user to search for specific values (e.g. strings, numbers, etc.). The set of addresses containing the specified values is returned and further searches may be performed on that set. The user of the memory scanning tool might use the target application in such a way that changes the value contained by the data structure or variable of interest. Subsequent searches for this modified value would then help isolate the desired address from the rest of the addresses returned by the initial search. Repeating these steps several times will often yield a small number of addresses that contain the targeted information.

While memory scanning provides a way to find the address of specific data at one point in time, it does not usually enable the address of that data to be found again in the future without repeating the process of memory scanning. Frequently, the data of interest is allocated dynamically on the heap, and as a result its address will usually change between instances of the program's execution, or even as the state of the application changes while running. In some cases, a variable may be stored in static memory, which is at a fixed offset from the base of the executable when it is loaded into memory (as is the case for objects of static storage duration in C and C++). As a result, finding the location of such a variable through memory scanning once can be sufficient to enable the location of that variable in the future. As detailed later, variables in static memory can also assist in locating variables stored in non-static memory, as they may contain pointers that directly or indirectly, via sequences of pointers and offsets, lead to the variables in non-static memory. Using sequences of pointers and offsets to traverse data structures from a predictable location to an unpredictable location containing desired data is a core concept explored by this paper.

To aid in understanding the problem solved by PSR, consider the following example for context before moving on:

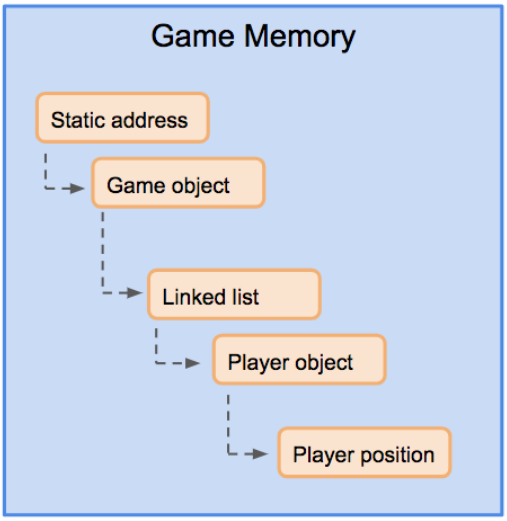


Figure 1: Objects in memory connected by a sequence of pointers.

The diagram represents an imaginary game application's process memory. The orange rectangles represent objects and data structures within the game, and the dotted arrows represent instances where one object contains a pointer to another object. Note that it is possible to start at a static address and follow a sequence of pointers to the data structure containing player position. Knowing the existence and details of this pointer sequence would be highly useful for the purpose of accessing player position at any given time.

Below, a more detailed diagram of a similar group of objects includes information about each of the objects' contents, including the locations of data members and pointers within each object.

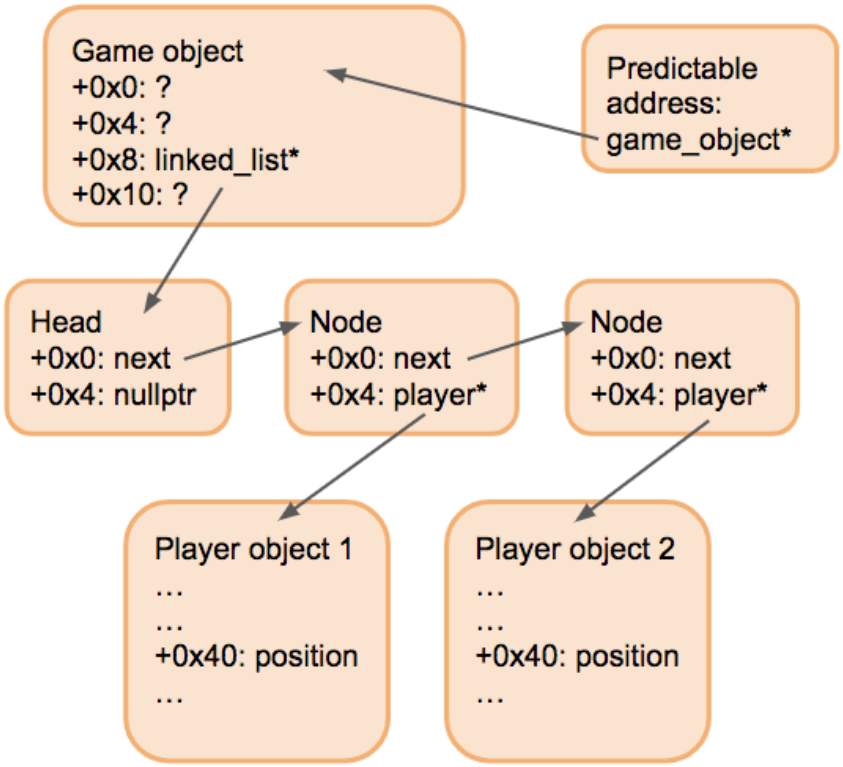


Figure 2: Objects in more detail, with internal offsets shown for various data members and pointers.

The pointer sequence leading from the predictable, static address shown above to the first player's position can be represented as follows:

```
[predictable address], [+0x8], [+0x0], [+0x4], +0x40
```

Where the [] operator means to dereference (i.e., take the value at the address inside the brackets), and the + sign means to add an offset of the value that follows. The commas separate each step of the sequence, and the result of each step is passed onto the next. In other words, the sequence can be read out as, "Dereference the predictable address, add 0x8 to the resulting value and dereference, add 0x0 to the resulting value and dereference, add 0x4 to the resulting value and dereference, then add 0x40 to the resulting value to get to the location of the player position data". This sequence traverses the objects in the diagram above to get to the first player's position information.

2 Prior Art

When memory scanning returns an unpredictable address for a particular variable or data structure, obtaining fast, reliable access to that data requires knowledge of a sequence of offsets and dereferences that lead from a predictable location in memory to the data of interest. Two previously existing methods for discovering such sequences are Cheat Engine's pointer scanner and manual dynamic reverse engineering.

2.1 Techniques

2.1.1 Automated Pointer Scanner

Cheat Engine's pointer scanner[1] attempts to identify sequences of offsets and dereferences through a brute force approach. Pointer scanner starts with predictable addresses (e.g. those located in static memory or early stack frames), and searches for sequences by dereferencing every value stored at those addresses as if they were pointers. The values pointed to by the first set of values are dereferenced in turn, and this process repeats, identifying sequences leading to a desired address. Along the way, values at offsets within a certain range from each value are also dereferenced, expanding the search tree and enabling the identification of sequences that involve offsets. The main problem with pointer scanner is its performance. It can take hours to identify sequences that are several pointer levels long. Pointer scanner also tends to produce large numbers of unstable sequences that do not persist between different runs of the application, requiring multiple iterations of scanning to isolate stable sequences.

2.1.2 Manual Reverse Engineering

Manual dynamic reverse engineering involves using a debugger to observe the execution of the target binary. The process typically starts with setting a memory breakpoint on the address containing the data of interest and then using the debugger to observe the instructions executed prior to hitting the memory breakpoint. In order to obtain the full picture of a sequence, many breakpoints are often required to be placed at locations prior to the data-accessing instruction. This is challenging because it can be difficult to place breakpoints in the correct locations. For example, some breakpoints are hit too frequently, where many of the hits are not relevant. Also, it can be hard to follow sequences when values are pushed onto and popped from the stack. Like Cheat Engine's pointer scanner, the main problem with the manual reverse engineering approach is the amount of time required to identify sequences.

2.2 Assessment

The manual debugging-based approach to deriving pointer sequences, while work-intensive, typically yields reproducible higher-value results than the brute force methods employed by pointer scanning tools such as Cheat Engine. However, the bulk of the tedious work involved can be broken down at a high level into a few core steps performed in repetition:

1. Get the register read during the execution of an instruction where the data of interest was accessed.
2. Find which previously executed instruction most recently set the value of that register.
3. Get the register read during that previously executed instruction.
4. Repeat, starting at step 2, until a previously executed instruction is encountered where no register was read from.

The manual approach to pointer sequence determination is therefore well-suited for automation.

[1] http://cheatengine.org/help/pointer-scan.htm

3 Pointer Sequence Reverser

Pointer Sequence Reverser (PSR) is a tool that implements and automates the manual approach. It takes the process ID of the target application and the address of the data as input, observes the execution of the process' code until the data is accessed, and then returns the set of instructions used by the process to access the data as output. PSR is written in C++ using the Win32 debugging API[2] and is implemented as two main components: a debugger and a tracer.

3.1 Design and Implementation

C++ on top of the Win32 debugging API was chosen after surveying other potential other approaches including Python and plugins for various debuggers. Python was ruled out due to its nature as a high-level language with less immediate access to the Win32 API and for performance concerns. Implementing it as a plugin for a debugger was also ruled out as existing ones either did not support x64 or have a suitable plugin API.

Finally, a core design goal was to focus on simplicity and workflow agility while avoiding over-engineering and complexity where feasible. PSR is intended to augment the work of a reverse engineer and, accordingly, it typically produces a number of incomplete sequences alongside more useful complete sequences, allowing the reverse engineer to choose which to analyze further.

3.2 The Debugger

The debugger attaches to a target application and sets a one-time memory breakpoint on the user supplied address. The debugger also repeatedly sets the trap flag on the CPU in each of the application's thread contexts. This causes execution control to return to the debugger after each instruction is executed, at which point the tracer is called to store the instruction. As the debugger has no understanding of the instructions' binary format, it sends bytes totaling the maximum potential byte sequence length for each recorded instruction. When the debugger detects that the memory breakpoint has been hit, it again calls the tracer to perform analysis of the recorded instructions. After analysis has completed, the debugger allows the target application to continue running and resets the memory breakpoint. The debugger waits a short amount of time between resuming process execution and resetting the memory breakpoint. This allows execution to advance past the memory breakpoint-triggering instruction before the memory breakpoint is set again. Otherwise, the target application would become stuck, infinitely executing the same instruction that triggered the access violation.

3.2.1 Win32 Debugging APIs

PSR uses the following Win32 functions to facilitate debugging:

- `DebugActiveProcess`: Used to attach to the target application and perform debugging operations on it.
- `DebugSetProcessKillOnExit`: Called with `FALSE` to disable termination of attached processes on exit.
- `VirtualQueryEx`: Used to obtain a page's protections before setting a memory breakpoint within it.
- `VirtualProtectEx`: Used to modify page protections in order to set memory breakpoints.
- `GetThreadContext`: Used to access information about a thread's context at a given point in time. For example, the value in the instruction pointer register when a memory breakpoint is hit.
- `SetThreadContext`: Used to set the trap flag on CPU to enable single stepping.
- `WaitForDebugEvent`: Used by PSR's main debugging loop to wait for a debug event to have occurred.
- `ContinueDebugEvent`: Allows the target application to resume execution after a debug event.

[2] https://msdn.microsoft.com/en-us/library/windows/desktop/ms679303(v=vs.85.aspx)

- `CreateToolhelp32Snapshot`: Used to enumerate the threads running in the target process.

3.2.2 Memory Breakpoints

PSR implements at-address memory breakpoints[3][4] by using `VirtualProtectEx` to set the `PAGE_GUARD` memory protection option on the target address. PSR's debugger component additionally stores the target address to identify specific memory accesses. When the target application attempts to access a page where `PAGE_GUARD` is applied, it raises a `STATUS_GUARD_PAGE_VIOLATION` that is passed to the debugger as a debug event. The debugger must then check whether the address the target application attempted to access matches any stored memory breakpoint addresses. If so, the debugger treats the event as a memory breakpoint having been hit and acts accordingly.

3.2.3 CPU trap flag

x86 and amd64 processors contain a FLAGS register (called EFLAGS and RFLAGS on x86 and amd64, respectively) that contains the current state of the processor. The bit at position 8 in the FLAGS register is the trap flag; when set it causes the CPU to execute one instruction and then stop. A single-step exception is sent to the debugger after the execution of each instruction, allowing the debugger the opportunity to record every instruction executed by the application. The trap flag must be reset by the debugger after each single-step exception is caught.

3.3 The Tracer

Each executed instruction is stored by the tracer in order. Instructions executed by each thread are stored separately. Alongside each instruction, the tracer also records which registers have been modified since the last instruction as well as their new values. It is possible to determine which registers have been modified by comparing the thread context from one instruction to the next. Only modifications to the general-purpose registers are recorded. The stack pointer and stack base pointer registers are also excluded from the record of modifications since they are not typically meaningfully involved in the traversal of data structures. This would complicate the process of analysis if included. When the memory breakpoint is finally hit, the tracer analyzes the recorded instructions and data.

During the analysis phase, the tracer uses Capstone[5] to disassemble and understand each recorded instruction. The first step in analysis is to determine the register used during the access of the specified data. For example, given the following instruction:

```
mov eax, [ecx + 8]
```

If this instruction triggered the memory breakpoint with a read, then the value in `ecx` was used to access the data, and it can be said that `ecx` has been read from.

Capstone enables the automatic identification of registers read from by instructions. Currently, PSR does not support automatic identification of registers written to during instructions, but could have an option enabling users to manually identify the appropriate register. For example, given the following instruction:

```
mov [ecx + 8], edi
```

If triggering the memory breakpoint with a write, the user would then be able to specify that `ecx` contained a value used to access the data.

Once the register used for the initial access to the data is determined, PSR then works its way back through the recorded instructions in an attempt to understand the origin of the value in the relevant register during the access. As mentioned previously, this is achieved through a series of steps. The steps below describe in

[3] http://waleedassar.blogspot.com/2012/11/defeating-memory-breakpoints.html
[4] http://www.codereversing.com/blog/archives/79
[5] http://www.capstone-engine.org/

detail how PSR performs the analysis of the recorded instructions.

1. Obtain the value of the target register for the identified instruction.

The value of a register when a given instruction is executed can be determined by examining the record of register modifications. As mentioned previously, the tracer stores the list of modified registers and their new values for each recorded instruction. Starting from the identified instruction, PSR scans this list for the last time the target register was modified. The value resulting from that modification is the value of the register when the identified instruction triggered the memory breakpoint.

2. Find the last time that value showed up in a trace of the recorded instructions.

Sometimes, a value can appear more recently in the trace than the time at which it was loaded into the previously identified register. For example, this would be the case if it was contained in another register. The last time the value appeared in the trace can be determined using a method similar to the one used in Step 1. Rather than focusing on the registers themselves, the search focuses on values set by register modifications. In particular, this step identifies relevant instructions by searching for values contained in registers instead of searching for the last time the read register was modified. The latter might seem to be a sensible approach, but it runs into obstacles when a register obtained its value by a pop from the stack, or some other operation where a stable sequence cannot be obtained.

Similarly, it might seem more efficient to search for the first time a value appears in the recorded instruction trace instead of the most recent. For example, this could yield a means to shortcut irrelevant instructions that simply shuffle a relevant value around. However, this approach could miss completed sequences residing between the first and most recent occurrence of a value. This could happen if instruction recording began while the target application was performing a sequence traversal. In such a situation, it would not be possible to identify all of the instructions completing a sequence until the partially completed sequence is purged from the trace; the instruction trace has a finite size, and once reached, the oldest instructions are overwritten. Furthermore, complete sequences identified could be incorrect if the data structures traversed by the target application occupy memory that was freed and reallocated between the first and last occurrence of the target value. In this case, the initial instance of the value – or values leading to it – may have been freed, and the most recent occurrence of the value is a separate instance within a newer allocation located at the same place in memory. This is an instance of the ABA problem.[6]

PSR uses Capstone to determine if trace instructions used stack registers (e.g. `esp` or `ebp`) to read the value. If so, the search continues backward through the trace until a non-stack register is read from. Reads from stack registers are ignored because sequences that traverse the stack are usually unstable due to the stack's inherent volatility.

3. Obtain the register read from for the instruction at that location in the trace. Capstone analyzes the instruction as before.

4. If the instruction performed a read from the instruction pointer or used an immediate value, the analysis is complete.
Else, return to Step 1 with the newly identified register/instruction pair.

Each of the trace instructions identified in Step 2 are considered relevant as they are part of the pointer sequence set used by the binary to access data at the originally specified target address. These instructions make up the bulk of PSR's output. PSR also performs additional processing to remove redundant instructions.

[6] https://en.wikipedia.org/wiki/ABA_problem

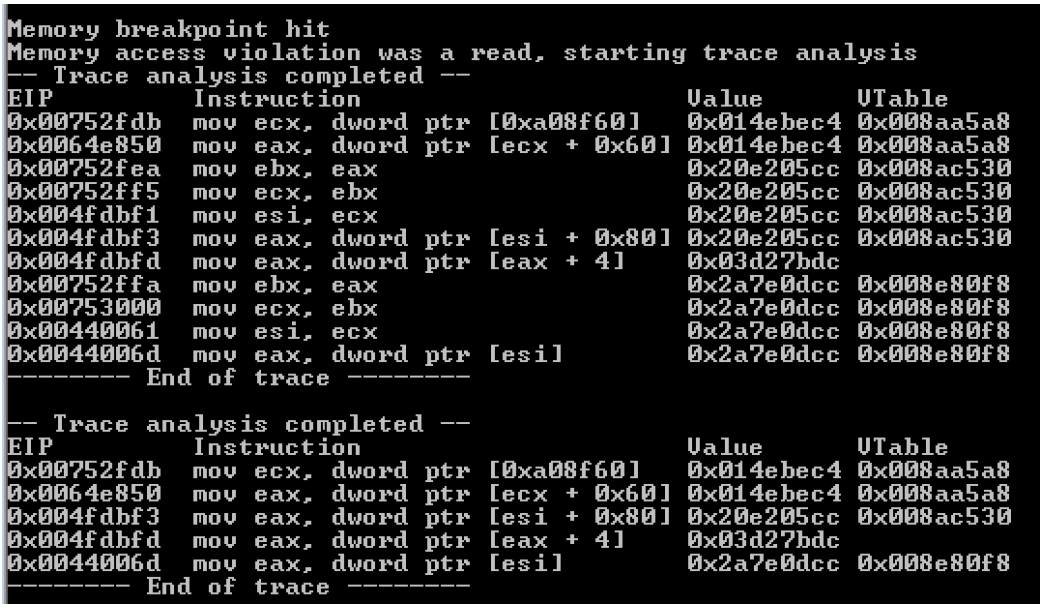


Figure 3: Example trace, before and after minimization to remove unimportant instructions.

3.4 Trace Minimization

Similarly to test case reductions performed by fuzzers to remove irrelevant inputs, PSR can remove redundant instructions to make pointer sequences concise and easier to understand. Redundant pointer sequence instructions are those that handle a relevant value, but do not dereference or add a useful offset to it. In many cases, it may be observed that a pair of redundant instructions simply move a relevant value from one register to another and then back again. Such instructions do not reveal any additional information about the pointer sequence and can make understanding the output harder.

In order to remove redundant instructions, PSR iterates through every relevant instruction in the list, starting with the earliest. It obtains the relevant value for each instruction and then looks for the most recent occurrence of that value in the list. If another occurrence is found, it removes everyinstruction between the first and last instructions containing the value, including the last one.

From the output, users can easily see a sequence of offsets and dereferences used by the binary to traverse a collection of its data structures. The most helpful sequences begin at a static address relative to the base of the executable. Such sequences are generally stable across application restarts. As a result, these sequences lend themselves to the applications mentioned previously, such as developing code that interoperates with the target application.

3.5 Value Column

PSR's output also includes a column for values. The Value column holds the values contained in the relevant register of each instruction. It acts as a visual aide, as it can be useful to know the exact addresses an application encountered while traversing its own memory. Often these addresses belonged to interesting data structures when PSR was run, and can afford users the opportunity to explore these structures in memory. For example, given the following recorded instruction:

```
mov eax, [ecx + 8]
```

If `ecx` had a value of 0x4A0C7F80 when the instruction was recorded, the Value column for this instruction will contain `0x4A0C7F80`.

3.6 Vtable Identification

PSR attempts to identify possible vtable pointers located at the addresses in the Value column and displays them in the Vtable column. C++ supports virtual inheritance, enabling derived classes to override the implementation of `virtual` methods of their base classes. This enables the most derived version of the method to be invoked when called on a base class pointer or reference. In general, as the compiler may not know the exact class of the polymorphic object a base class pointer or reference refers to, each polymorphic object will contain a pointer to a metadata table describing its class. These tables, vtables, may contain runtime type information, but primarily contain function pointers for each virtual method used in the derived class. The pointers to this table that exist within object instances are called vtable pointers. Vtable pointers are placed at the beginning of object structures in memory. In this way, vtables are used to facilitate the run-time method binding required for objects with virtual methods. This solution is not tied to C++ itself, but is commonly used by compilers to implement dynamic dispatch,[7] including the Microsoft C++ compiler used by Visual Studio, Clang, and GCC.

Further information on the layout of vtables and polymorphic objects in memory can be found at:

- http://www.openrce.org/articles/full_view/23
- https://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf
- http://syssec.rub.de/media/emma/veroeffentlichungen/2016/12/22/marx_ndss2017.pdf
- https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2017/may/assethook-a-redirector-for-android-asset-files-using-old-dogs-and-modern-tricks/

Since vtables are associated with polymorphic objects, the presence of a vtable pointer at an address in the Value column indicates the presence of an object, and also that the object is somehow included in the sequence. Information about vtable pointers can help a reverse engineer with prior knowledge about a target application place PSR's output in context. For an example, see Section 4 on page 13.

PSR attempts to identify vtable pointers using a heuristic technique involving page protections. First, it dereferences each address in the Value column to determine the value contained at the address. The resulting value could be a vtable pointer, and to find out, PSR observes the memory protections applied at the address it points to. Since vtables are frequently stored in read-only memory, the memory protections at the address should be read-only. If this is the case, PSR checks the protections of the memory pointed to by the first entry in the potential vtable. Since vtables contain function pointers, and function pointers point to executable code, the memory pointed to should have executable page protection. If this is also the case, then PSR assumes that it has identified a vtable and includes its address in the Vtable column of the instruction.

3.7 Support for x86, x64, and Position-independent Code

PSR was originally developed to support non-position-independent executables on x86. Adding support for position-independent executables and x64 executables required a small amount of additional work. In order to support 64-bit applications, it was necessary to convert many strings referring to x86 register names in the source code to their amd64 counterparts. For example, all instances of "Eax" were to be changed to "Rax." Note that this is not a perfect solution however, and does not account for instances where 32-bit registers are used in 64-bit code. In order to support position independent code, it was necessary to modify PSR so that instruction analysis stops when the instruction pointer is read from, as in position-independent executables, certain offsets are added to the instruction pointer to access data in predictable instruction address-relative

[7] https://en.wikipedia.org/wiki/Dynamic_dispatch

locations.

4 Case Study

This section presents a full use case example of PSR by applying it to a popular first-person shooter PC game written in C++. We use PSR to identify a pointer sequence leading to player objects. Accessing player object data at runtime can enable development of bots, or provide a player with additional or enhanced game information. The game studied here is a 32-bit, non-position-independent binary.

4.1 Locating Target Data

Before PSR can be used, the address of the data of interest must be determined. In this case, the data of interest is the player object. The game studied in this section features a single-player mode using computer-controlled player bots that will be used for testing. Assuming that player objects are managed in a similar way for both human and computer players, identifying a pointer sequence leading to a user-controlled player's object should work for the players controlled by the computer as well.

To obtain the address of the user-controlled player's object at a particular point in time we will perform memory scanning. Using Cheat Engine's memory scanning functionality, we can locate the user-controlled player's object indirectly by searching for one of its data members. In the target game, a number of viable techniques exist for locating a data member within player objects. One of the simplest is to switch the player's active weapon and look for corresponding changes in memory. Changing the active weapon and scanning for 4-byte integers containing those numbers quickly isolates one address.

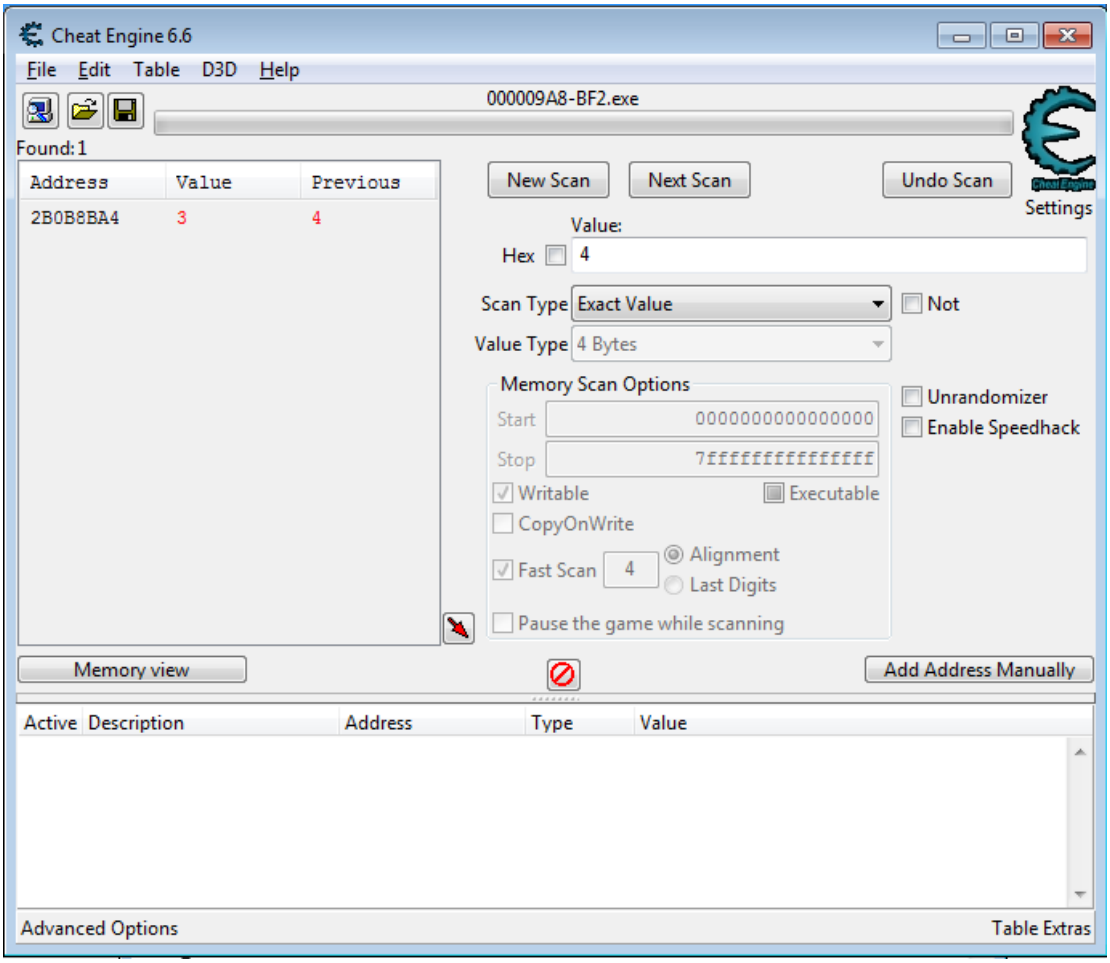


Figure 4: The address of the user-controlled player's weapon selection at a particular point in time.

4.2 Obtaining a Suitable Address

Using OllyDbg,[8] a memory breakpoint can be set on the address containing the weapon selection value. One of the breakpoint triggering instructions is `mov eax, [esi + 1e0]`, suggesting that the weapon selection could be stored at an offset of 0x1e0 within its containing object. A quick check of the memory using Cheat Engine's structure dissect tool at the address held by `esi` shows that the address appears to have a vtable pointer, suggesting that it is indeed the beginning of an object.

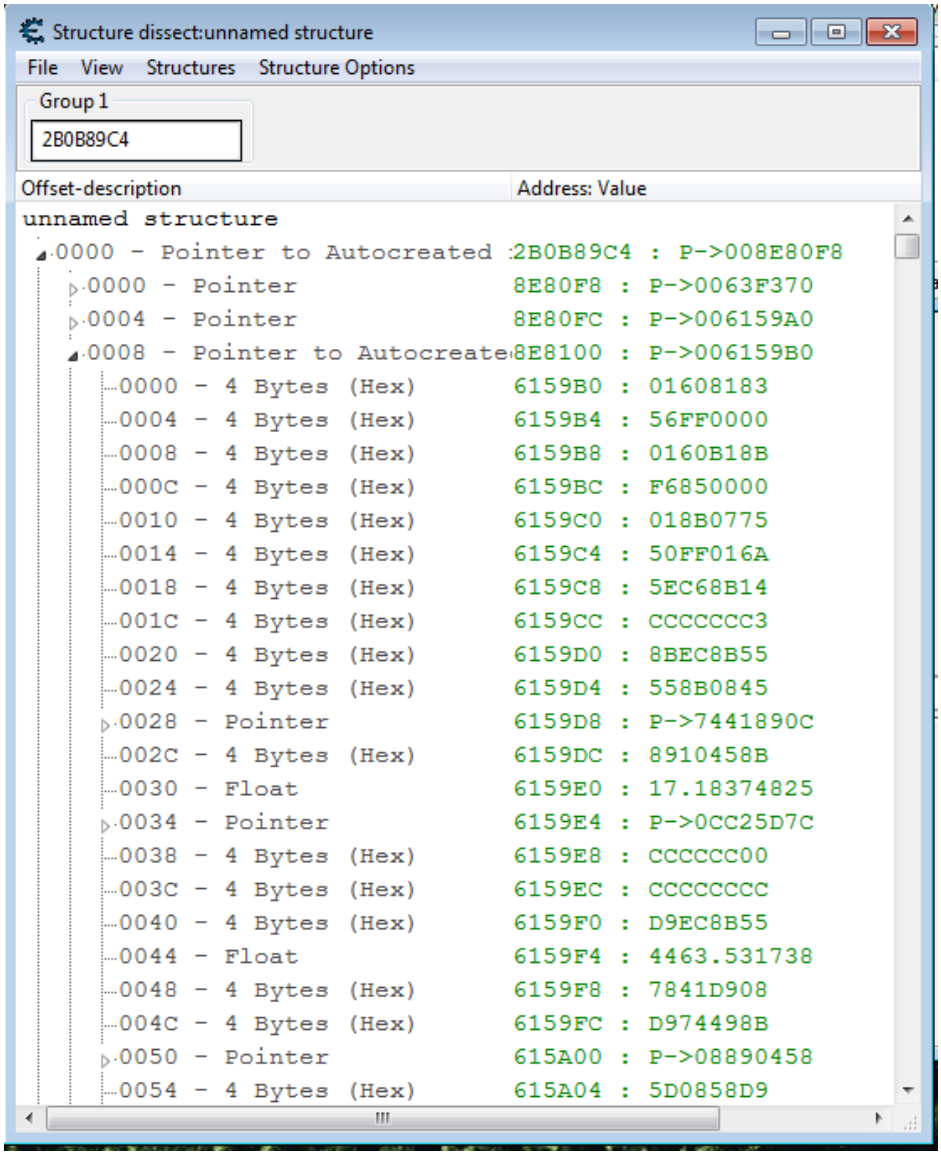


Figure 5: The bytes shown correspond to plausible assembly code. Most apparently, the many hexadecimal CCs correspond to INT3 instructions which are commonly found between chunks of actual code.

By supplying PSR with the address of the containing object instead of the weapon selection value, it can return results faster as the object is accessed more frequently than the weapon selection value. A few minutes after the game's process ID and the address of the object are supplied to PSR, several traces are produced. Many of the traces produced are fairly unhelpful – they do not start with a static address – but several of the traces do begin with a static address.

[8] http://www.ollydbg.de/

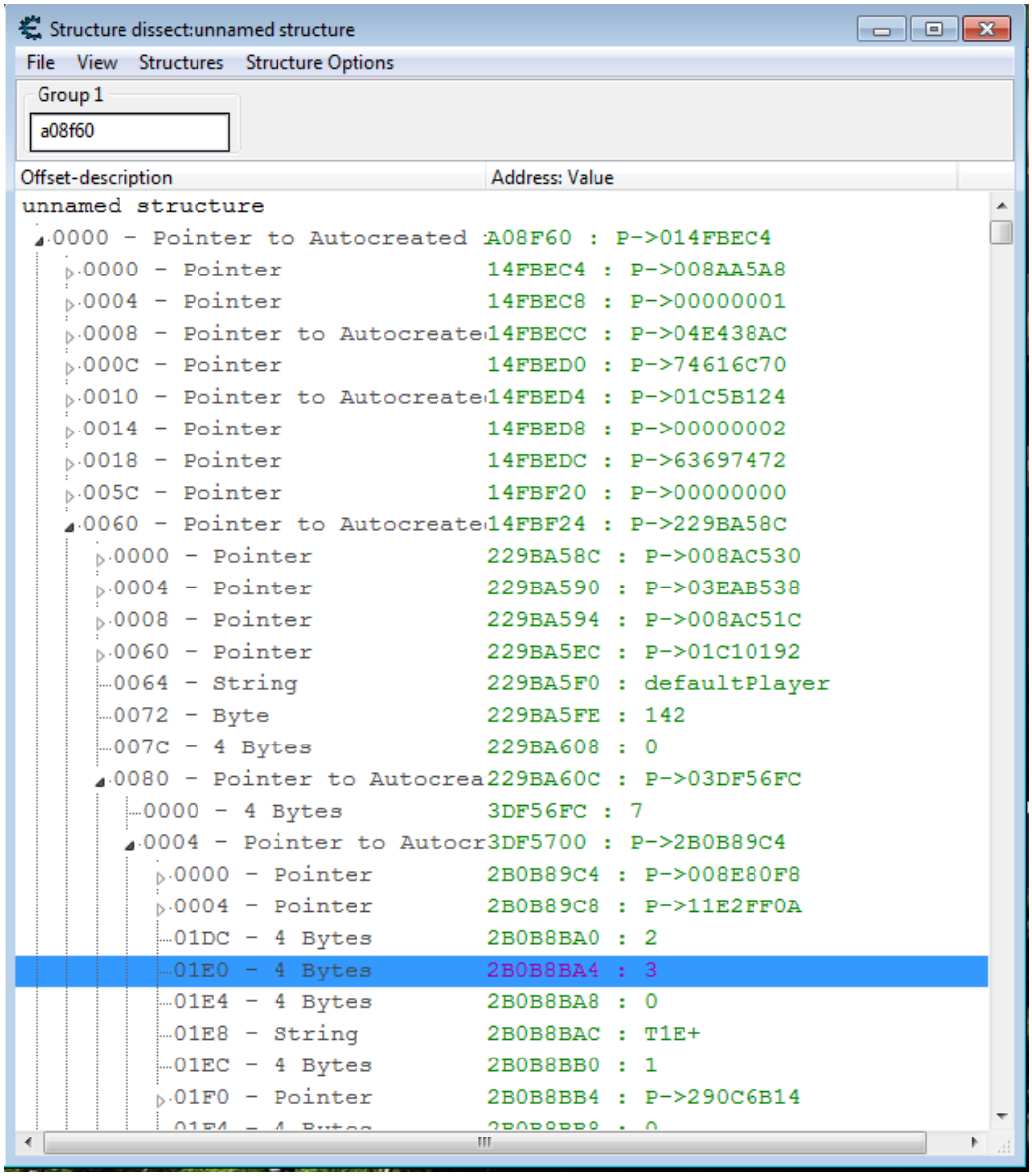
4.3 Analyzing PSR Output and Identifying Structures

Many of the traces appear to show the same few sequences. One of the common traces shows a sequence leading from an unchanging address (`0xA08F60`) all the way to the object containing player's weapon selection. The below screenshots show the sequence and the memory associated with it.

Note: Some offsets have been omitted for space.

```
-- Trace analysis completed --
EIP         Instruction                          Value      VTable
0x00752fdb  mov ecx, dword ptr [0xa08f60]        0x014fbec4 0x008aa5a8
0x0064e850  mov eax, dword ptr [ecx + 0x60] 0x014fbec4 0x008aa5a8
0x004fdbf3  mov eax, dword ptr [esi + 0x80] 0x229ba58c 0x008ac530
0x004fdbfd  mov eax, dword ptr [eax + 4]    0x03df56fc
0x0044006d  mov eax, dword ptr [esi]        0x2b0b89c4 0x008e80f8
-------- End of trace --------
```

Figure 6: A trace from a predictable address to an object containing the weapon selection value.



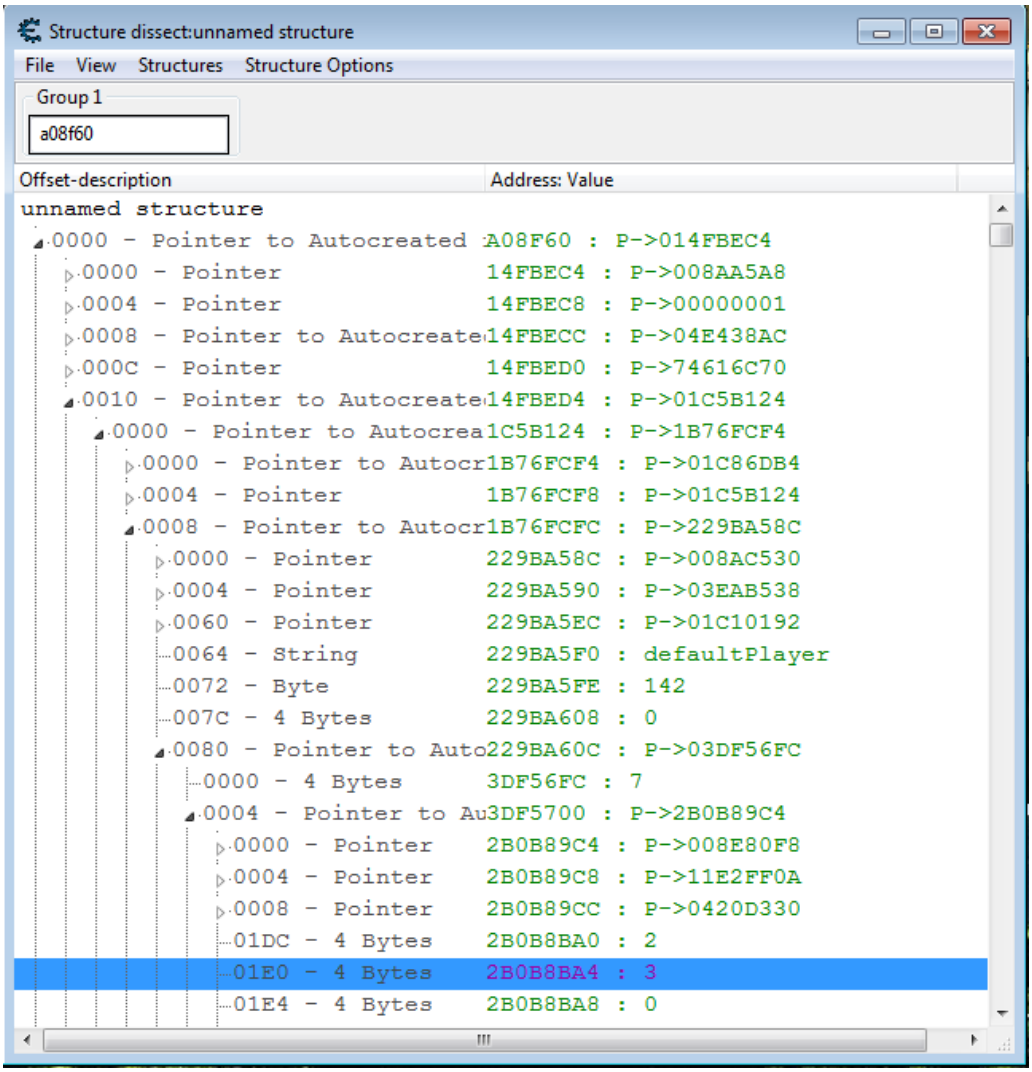
The pointer sequence of offsets and dereferences starts at `0xA08F60` and leads to the weapon selection (`0x3`) value at address `0x2B0B8BA4`. The address supplied to PSR, the address of the object containing the weapon selection supplied to PSR is naturally present as well. Also, observe that the user-controlled player's name, "defaultPlayer", is present within the object starting at `0x229BA58C`.

Another common trace in PSR's output uses a different pointer sequence leading to the object containing the weapon selection value. The below screenshots show this sequence and associated memory contents.

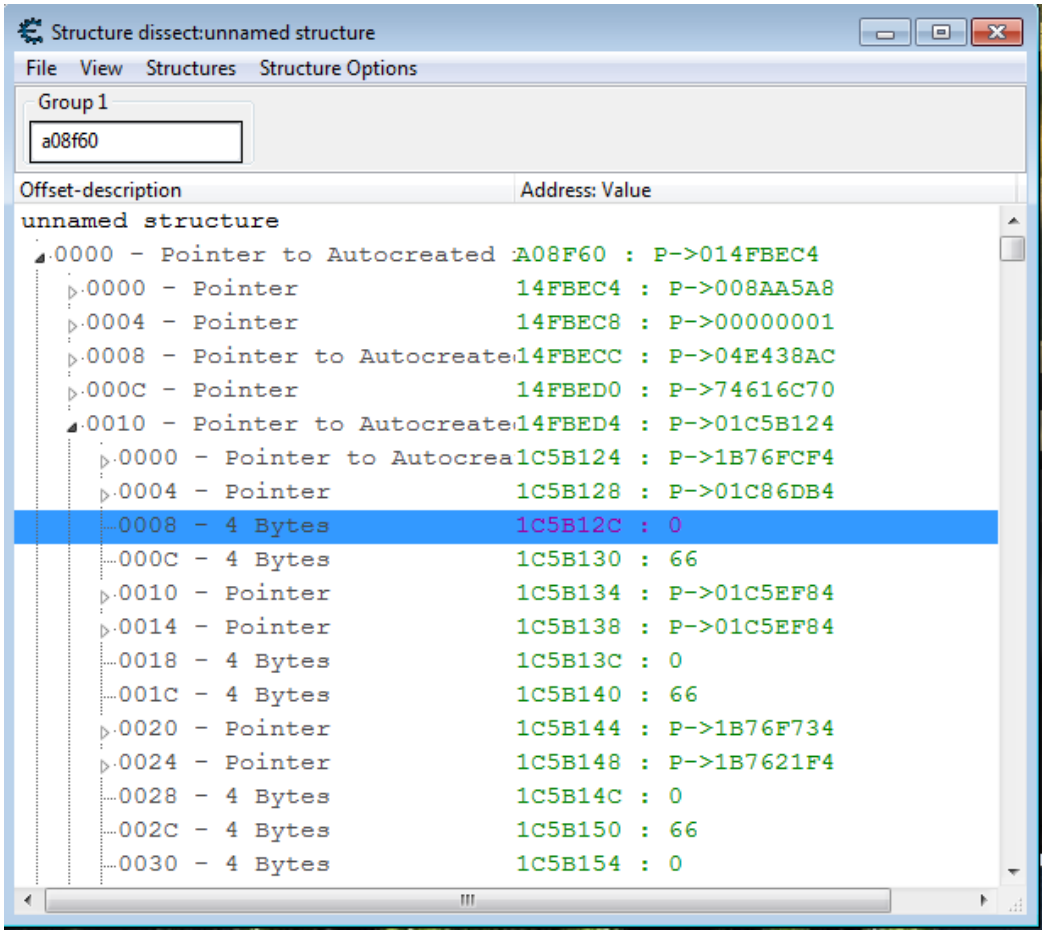
Note: Some offsets have been omitted for space.

```
-- Trace analysis completed --
EIP         Instruction                          Value      VTable
0x004dc9d9  mov ecx, dword ptr [0xa08f60]   0x014fbec4 0x008aa5a8
0x0065ddd0  lea eax, [ecx + 0xc]            0x014fbec4 0x008aa5a8
0x004dc9ec  mov eax, dword ptr [eax + 4]    0x014fbed0
0x004dc9ef  mov edi, dword ptr [eax]        0x01c5b124
0x004dca00  mov ecx, dword ptr [edi + 8]    0x1b76fcf4
0x004fdbf3  mov eax, dword ptr [esi + 0x80] 0x229ba58c 0x008ac530
0x004fdbfd  mov eax, dword ptr [eax + 4]    0x03df56fc
0x005e6f7f  mov edx, dword ptr [eax]        0x2b0b89c4 0x008e80f8
-------- End of trace --------
```

Figure 7: Alternate sequence from a predictable address to an object containing the weapon selection value.



Further analysis of the data structure at address `0x1C5B124` reveals that it is a circular doubly-linked list containing pointers to every player object. The screenshot below shows the sentinel node of the list.



As can be seen above, the pointers at offsets `0x0` and `0x4` point to the first and last nodes. These each contain pointers to the first and last player objects, respectively. The null pointer further at offset `0x8` indicates that it is the sentinel node of the list.

Examining the pointers contained within each node confirms that they encapsulate player data. The objects they point to contain the names of the two players currently playing, the user-controlled "defaultPlayer," and the computer-controlled "D. Yee." Also observable within the following screenshots are the vtable pointers of these player objects; they are the same for both objects (`0x8AC530`), indicating that they are of the same type and not different subclasses of a shared base class.

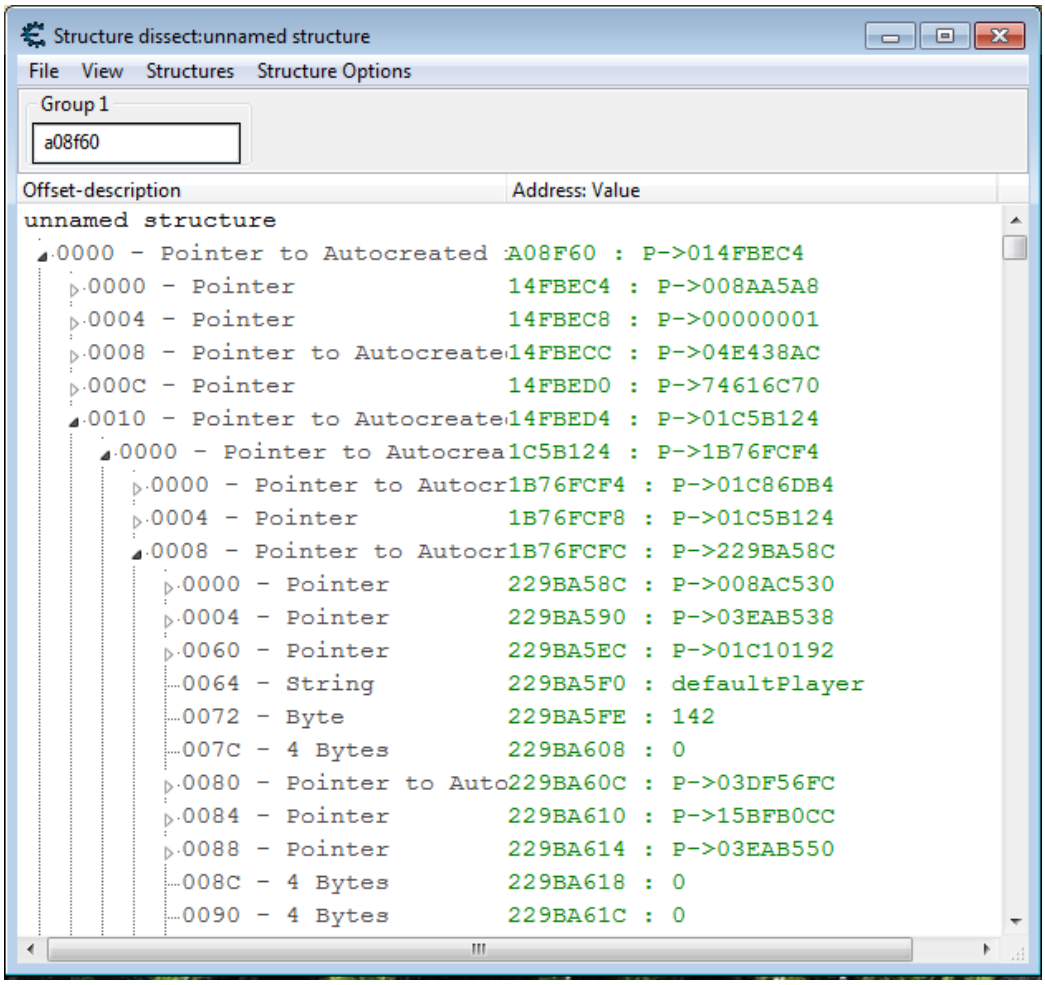


Figure 8: Navigating to the player object for "defaultPlayer" through the linked list.

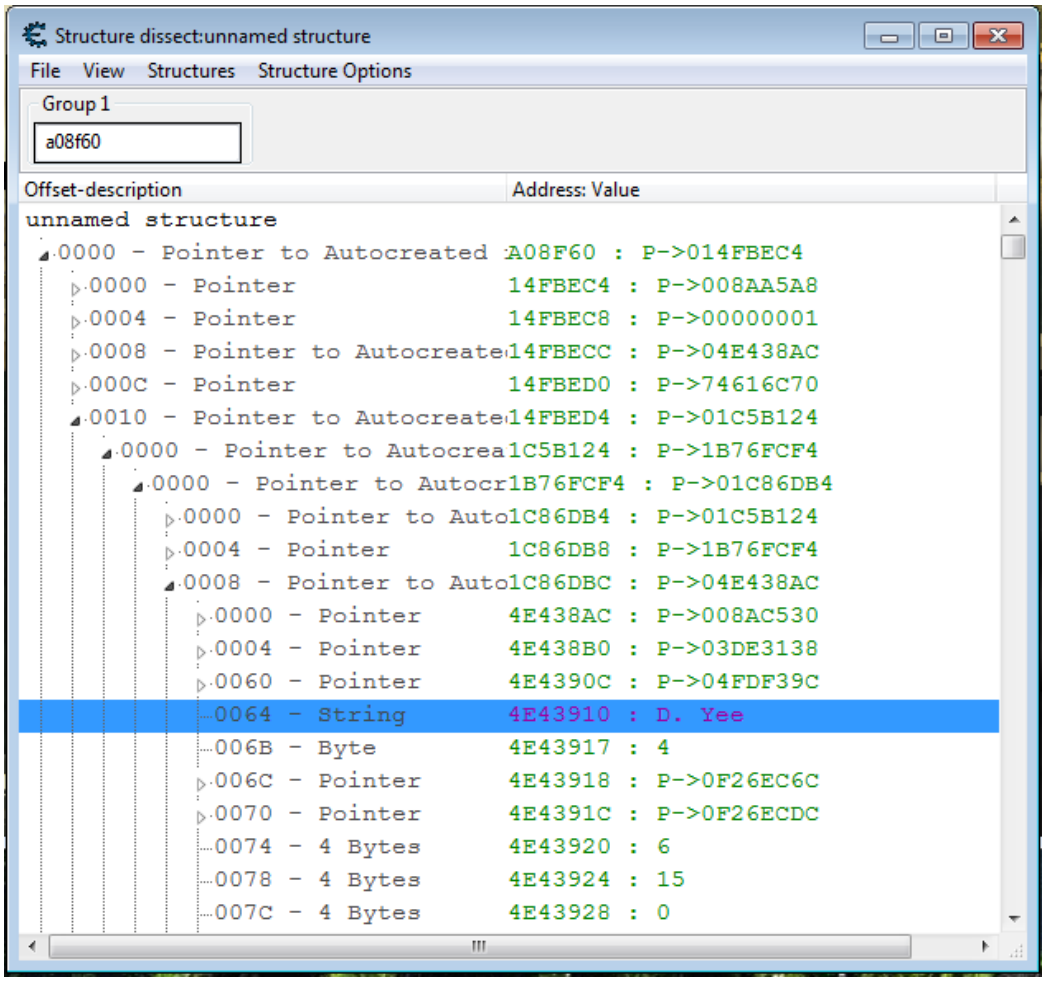


Figure 9: Navigating to the player object for "D. Yee" through the linked list.

Given the sequence produced by PSR, it is now possible to quickly access any player's object in the game at any time in the future, enabling mods or other programs to interoperate with this game. Additional reverse engineering, with PSR or other tools, could be performed to identify other interesting data structures or determine more about the structure of the player objects and the information contained within. Lastly, now that it is known that the player object vtable resides at `0x8AC530`, context can be given to traces produced by PSR in the future when `0x8AC530` shows up in the Vtable column. Such occurrences would indicate that the pointer sequence deals with player objects.

5 The End

5.1 Conclusion

This paper discussed and demonstrated an implementation of a technique for obtaining reliable access to data structures within a binary. Automated pointer sequence reversing is faster, and usually more accurate, than Cheat Engine's pointer scanner, a similar tool implementing a different technique. It provides a starting point for analyzing an application's data structures, aiding analysis in order to gain a deeper understanding of an application and how its internal data structures relate to one another.

PSR does not produce perfect results 100% of the time, but instead quickly produces a number of results of varying quality, allowing the user to select the best output. Although PSR readily lends itself to reverse engineering for the purpose of developing game mods, it may be useful in other scenarios as well. The code for PSR can be found at https://github.com/nccgroup/psr.

5.2 Acknowledgements

I would like to thank my editor, Jeff Dileo, for his hard work, valuable time, and keen insight while I developed PSR and wrote this paper. I would also like to thank NCC Group for encouraging research projects like this one.