



<http://www.nologin.org>

# Reverse Engineering: Memory Analysis

---

skape  
mmiller@hick.org

*Last modified: 12/29/2003*

# Contents

<b>1</b>	<b>Foreword</b>	<b>3</b>
<b>2</b>	<b>Introduction</b>	<b>4</b>
<b>3</b>	<b>Process Memory Analysis</b>	<b>6</b>
3.1	Accessing Process Memory . . . . .	6
3.1.1	ptrace . . . . .	6
3.1.2	procfv . . . . .	8
3.1.3	win32 . . . . .	11
3.2	Predicting Data Structure Layout . . . . .	13
3.3	Locating Data Structures . . . . .	14
3.3.1	Known Value . . . . .	14
3.3.2	Contiguous Patterns . . . . .	15
3.3.3	Heap Enumeration . . . . .	16
3.3.4	Referential Address Tree . . . . .	16
3.4	Identifying Data Structure Attribute Types . . . . .	17
3.4.1	Integers . . . . .	17
3.4.2	Strings . . . . .	19
3.4.3	Pointers . . . . .	20
3.4.4	Other . . . . .	20
<b>4</b>	<b>Case Study: adom</b>	<b>22</b>
4.1	Setting Character Attribute Levels . . . . .	23
4.2	Setting Character Experience . . . . .	25
4.3	Setting Character Skill Levels . . . . .	27
4.4	Changing Item Attributes . . . . .	36
4.4.1	Type . . . . .	46
4.4.2	Count . . . . .	46
4.4.3	Weight . . . . .	47
4.4.4	Dice . . . . .	48
4.5	Enumerating MOBv . . . . .	50
4.6	Enumerating Items . . . . .	52
<b>5</b>	<b>Conclusion</b>	<b>54</b>

<b>A Case Study Source Code</b>	<b>55</b>
A.1 Setting Character Attribute Levels . . . . .	55
A.2 Setting Character Experience . . . . .	56
A.3 Setting Skill Levels . . . . .	57
A.4 Changing Item Attributes . . . . .	59
A.4.1 Type . . . . .	59
A.4.2 Count . . . . .	61
A.4.3 Weight . . . . .	64
A.4.4 Dice . . . . .	66
A.5 Enumerating MOBs . . . . .	69
A.6 Enumerating Items . . . . .	71

# Chapter 1

## Foreword

**Disclaimer:** This document was written in the interest of education. Some or all of the contents of this document may not be compatible with that of the reader's with regards to ethics and code of conduct as it pertains to reverse engineering. If the reader thinks that he or she might fall under this category, please do not read this document. This document does not attempt to address or even discuss the matter of ethics barring this disclaimer, and as such the author would request that any such comments be left to one's self. With that, as with all things that are free, this document and the code in it come with absolutely no warranty, as it were. The author cannot be held responsible for how any of the concepts or code in this document are used.

The case study presented in this document was done on the rogue-like role playing game (RPG) called **Ancient Domains of Mystery**, or **adom** for short. The version of **adom** analyzed was **1.1.1**. **adom** is a single player game. For more information about **adom**, please visit their website at <http://www.adom.de>.

The version of **memgrep** used in this document was **0.8.0**. **memgrep** can be downloaded from <http://www.hick.org/code/skape/memgrep>.

The author would like to thank **trew** ([trew@exploit.us](mailto:trew@exploit.us)) for brainstorming with the author and being an all around cool guy. Thanks also go out to **thief** ([thief@exploit.us](mailto:thief@exploit.us)) for coming up with the concept for **memgrep** and for brainstorming with the author on multitudes of topics. Finally, the author would like to thank all the people out there who are motivated to learn for their own satisfaction.

With that, on with the show...

## Chapter 2

# Introduction

Trying to analyze a black box can sometimes seem quite the daunting task. Whether the focus be on analyzing the functionality of a given application or on analyzing its internal structures, one is not likely to get far without the proper resources. This document attempts to take the reader through the basic methodologies of accessing process memory on various platforms and then proceeds to build on this platform by bringing about techniques that can be used to understand and decode internal data structures of an application. It is the hope of the author that the reader will walk away with a refreshed or better sense of understanding as it pertains to being able to reverse engineer the internals, at least in part, of an application based solely on the process' memory layout.

What good does accessing process memory do for a would-be reverse engineer? It really depends on one's goals and the application that is being approached. Memory analysis is useful for times when one is trying to figure out the relative security of an application without taking hours and hours to disassemble it. It is also useful for times when one wants to attempt to make the program do something it should not, or would not otherwise allow for. Some of the common scenarios where memory analysis can be very useful are things like:

1. Trying to find static encryption or authentication keys.
2. Trying to manipulate the state of a client application that does not have proper server validation.
3. Trying to cheat an application by modifying process memory to achieve a desired effect.

These broad categories can be applied to applications that fall into many dif-

ferent genres such as security software, internet software, and computer games.

The most common technique used in the field of reverse engineering at the time of this writing is disassembly analysis. This technique involves disassembling a binary into the assembly of the architecture that it was meant to run on. From there one can proceed to attempt to analyze how the program works. Though this technique is very powerful, one should note that it's also very tedious and time consuming for the person doing the analysis. A person tasked with doing analysis on an application will be required to be well versed in the assembly of the architecture that the application is meant to run on and must also be familiar with the architecture itself on a programmer's level. While there exist tools to help make this process easier for the person doing the analysis, disassembly analysis has far too many negative aspects to be considered accessible enough to the majority of those who might otherwise be interested in the field.

Enter memory analysis, stage left. This technique, though more limited, allows for a person tasked with reverse engineering to get a feel for how a program is written by inspecting the application's memory layout while it's running. Also, unlike disassembly analysis, memory analysis eliminates the requirement that the person be forced to be well versed in the assembly of the architecture that the application was written for. In place of this, all that the person has to be familiar with are the process address space layout principals as they apply to the given architecture. It goes without saying that the latter is far less a daunting task than the former. Indeed, memory analysis is more limited in scope due to the fact that one cannot analyze the algorithms and code structure of a given application, but instead one is able to analyze the layout and usage of data structures that are internal to the application.

When deciding what technique one should use, keep the following in mind. If one is attempting to do a security analysis of an application or is trying to make the program do something it shouldn't or didn't expect to without modifying the binary itself, memory analysis should be considered. If one is attempting to do general binary analysis in order to understand how a piece of software works or interacts, disassembly analysis should be considered.

Pressing forward, the reader should be expected to have a conceptual understanding of process memory layout with relation to binary sections (such as rodata, data, bss). Aside from this, all else that is required is the desire to learn or refresh.

## Chapter 3

# Process Memory Analysis

### 3.1 Accessing Process Memory

The ability to access a process' address space from the context of another process that is running in user-mode is a very powerful feature. This ability allows one to be able to perform real time analysis and debugging on another process without having to have launched the process with any special flags or parameters. Most popular operating systems expose an interface by which one can “open” or “trace” another process and in so doing allow for the calling process to be able to read from and write to arbitrary locations in memory, among other things.

This section will provide a basic description of the API's and methods used to read and write to a process' address space. Though the principals are the same across most platforms, the implementations are quite different. Each platform will tend to implement at least four logical operations: open, read, write, close. The open operation is used to gain access to another process and is always the first step before other operations can be performed. The read and write operations are used to read and write to a process' address space, respectively. Finally, the close operation is used to relinquish access when one is done operating on a process' address space and free up any resources that were allocated while being opened.

#### 3.1.1 ptrace

ptrace, short for process trace, is a UNIX specific interface to tracing a given process from the context of another process. ptrace is a standard from BSD 4.3 and SVr4. Most current UNIX derivatives have some form of ptrace. It is with this API that one can implement a mostly portable, at least across the UNIX

platforms, method of reading and writing process memory. The function one uses to perform these actions is the `ptrace` function which is prototyped as:

```
long int ptrace(enum __ptrace_request request, pid_t pid, void *
                addr, void * data);
```

The `request` parameter is used to tell `ptrace`, and thus the kernel, what action is to be performed. There are generally around 10 to 15 requests that can be performed but due to the focus of this document only four will be discussed. The names of these methods may vary from implementation to implementation, but the request operation is always the same<sup>1</sup>:

1. `PTRACE_ATTACH` (*open*)

**Synopsis:** This request allows the caller to attach to a process. This request must be sent before any other requests can be sent that will operate on a given `pid`. One can consider this the “open” or “trace” request. This request will stop the process specified by the `pid` argument with `SIGSTOP`.

**Arguments:** The `pid` argument should be set to the process id that one is going to attach to. The `addr` and `data` parameters are ignored and should both be 0.

**Return:** On success, zero will be returned. Otherwise, -1 will be returned and the error will be set in `errno`.

2. `PTRACE_PEEKDATA` (*read*)

**Synopsis:** This request allows the caller to read four bytes of data from an arbitrary address in another process’ address space.

**Arguments:** The `pid` argument should be set to the process id that is being operated on and has already been `PTRACE_ATTACH`’d successfully. The `addr` argument should be set to the address that is to be read from. The `data` argument is ignored and should be set to zero.

**Return:** On success, the value read from the location in memory will be returned. Otherwise, -1 will be returned and `errno` will be non-zero. One must be sure to verify that `errno` is set when -1 is returned, otherwise it’s possible that the -1 was actually read from memory and is not actually an error.

3. `PTRACE_POKEDATA` (*write*)

**Synopsis:** This request allows the caller to write four bytes of data to an arbitrary address in another process’ address space.

**Arguments:** The `pid` argument should be set to the process id that is being operated on and has already been `PTRACE_ATTACH`’d successfully. The `addr` argument should be set to the address that is to be written to.

---

<sup>1</sup>FreeBSD prefixes requests with `PT` instead of `PTRACE`, for instance.



The **data** argument should be set to a word value that is to be written to the address specified in **addr**.

**Return:** On success, zero is returned. Otherwise, -1 will be returned and the error will be set in **errno**.

#### 4. PTRACE\_DETACH (*close*)

**Synopsis:** This request restores a previously attached process to its running state and unsets the trace flag. No further ptrace requests, barring PTRACE\_ATTACH, can be sent after this point.

**Arguments:** The **pid** argument should be set to the process id that is to be detached from. The **addr** and **data** arguments should both be 0.

**Return:** On success, zero is returned. Otherwise, -1 will be returned and the error will be to in **errno**.

The only major limitation that ptrace has is that only a small, architecture dependent, amount of memory can be read and written at once. In order to emulate the ability to read and write more memory, one must wrapper the logical read and write operations that in turn call ptrace as many times as is necessary for all the bytes requested to be read or written.

This method is the preferred method on UNIX as it is the most portable. However, some UNIX variants require use of another interface: **procfs**.

### 3.1.2 procfs

procfs is, to put it simply, a virtual file system that allows for one to view the state of all the running processes by merely browsing a mounted directory on the file system. Some UNIX variants have added other features to their implementations of procfs, but the focus still remains on one being able to gather information about the state of running processes.

Most UNIX variants will mount the proc filesystem under **/proc**. For instance, the Solaris **/proc** directory will tend to look something like this:

```
$ ls /proc
0      1      1007  109   111   112   121   129   130   133
2      3      88    985   989
$ ls /proc/109/
as      cred      cwd      lpsinfo  lusage   map
pagedata rmap     sigact   usage    xmap     auxv
ctl     fd       lstatus  lwp      object   psinfo
root    status   watch
```

The remainder of this description will explain how one can make use of the `procfs` API to read and write process memory on Solaris. Other UNIX variants may have a similar interface, though it is not guaranteed to be the same. The four major operations are implemented as follows:

1. `open` (*open*)

**Prototype:**

```
int open(const char *path, int oflag, ...);
```

**Synopsis:** In order to attach to a process in Solaris using the `procfs` interface, one must use the `open` function on two virtual files in the `procfs` tree. The first file will be found under `/proc/[pid]/ctl`. This file, commonly referred to as the “control” file, can only be opened for writing and is used to control the state and behavior of the process that it’s related to. In this case it is used to stop and start the process when necessary. The second will be found under `/proc/[pid]/as`. This file, commonly referred to as the “address space” file, can be opened for read/write and is used to represent the address space of the process it’s related to. The file descriptor returned by `open` on this file will be used to read and write to the address space of the process.

**Arguments:** For the control file, the `path` argument should be set to `/proc/[pid]/ctl` where `pid` is the identifier of the process being attached to. The `oflag` argument should be set to `O_WRONLY`. For the address space file, the `path` argument should be set to `/proc/[pid]/as` where `pid` is the identifier of the process being attached to. The `oflag` argument should be set to `O_RDWR`, or more restrictive depending on one’s purposes.

**Return:** On success, both calls to `open` will return a non-negative integer that represents the descriptor that will be used to operate on the files. Otherwise, a negative value will be returned and `errno` will be set accordingly.

2. `pread` (*read*)

**Prototype:**

```
ssize_t pread(int fildes, void *buf, size_t nbyte, off_t  
             offset);
```

**Synopsis:** This function allows the caller to read memory from the address space of another process into a buffer local to the calling process. Unlike `ptrace`, this method is not limited to a static length and thus allows for more flexible and efficient reads.

**Arguments:** The `fildes` argument should be set to the value that was returned from calling `open` on the “address space” file. The `buf` argument should be set to a pointer to a local buffer that will hold the memory that

is read. The `nbyte` argument should hold the number of bytes that are to be read. Finally, the `offset` argument should hold the virtual memory address relative to the process being operated upon.

**Return:** On success, a non-negative integer will be returned reflecting the number of bytes actually read. Otherwise, a negative value will be returned and `errno` will be set accordingly.

3. `pwrite` (*write*)

**Prototype:**

```
ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
               off_t offset);
```

**Synopsis:** This function allows the caller to write to memory in the address space of another process from a buffer local to the calling process. Unlike `ptrace`, this method is not limited to a static length of bytes that can be written and thus allows for more flexible and efficient writes.

**Arguments:** The `fildes` argument should be set to the value that was returned from calling `open` on the “address space” file. The `buf` argument should be set to the pointer that holds the data that is to be written. The `nbyte` argument should be set to the number of bytes that are to be written. Finally, the `offset` argument should be set to the virtual memory address relative to the process being operated on upon.

**Return:** On success, a non-negative integer will be returned reflecting the number of bytes actually written. Otherwise, a negative value will be returned and `errno` will be set accordingly.

4. `close` (*close*)

**Prototype:**

```
int close(int fildes);
```

**Synopsis:** This function allows the caller to detach from a process’ control and address space interfaces.

**Arguments:** The `fildes` argument should be set to the value returned from calling `open` on the “control” file. Secondly, one should call this function with the `fildes` argument set to the value returned from calling `open` on the “address space” file.

**Return:** On success, zero is returned. Otherwise, -1 is returned and `errno` is set accordingly.

From a programming perspective, Solaris’ `procfs` interface is much cleaner and more robust than the POSIX `ptrace` interface<sup>2</sup>. The author is not aware of any significant disadvantages to using the `procfs` interface.

---

<sup>2</sup>Solaris also exports a `ptrace` interface.

### 3.1.3 win32

The Windows side of the house does not implement either a POSIX ptrace interface nor a procfs interface. Rather, operations on another process' address space are done through a set of functions exported in `kernel32.dll`. These functions closely resemble the procfs interface in principal. The interface described below is compatible with both 9x and NT-based versions of Windows.

#### 1. OpenProcess (*open*)

**Prototype:**

```
HANDLE OpenProcess(  
    DWORD dwDesiredAccess,  
    BOOL bInheritHandle,  
    DWORD dwProcessId  
);
```

**Synopsis:** This function allows the caller to open a process for a requested set of operations specified in the `dwDesiredAccess` argument. For the case of reading and writing process memory, one will want to specify the access flags `PROCESS_VM_READ` and `PROCESS_VM_WRITE`. The handle returned by this function will be used in subsequent calls by the read, write, and close functions.

**Arguments:** The `dwDesiredAccess` argument should be set to `PROCESS_VM_READ` and `PROCESS_VM_WRITE` bitwise OR'd together. The `bInheritHandle` flag should be set to `FALSE` as handle inheritance is not needed. Finally, the `dwProcessId` should be set to the identifier of the process that is to be operated upon.

**Return:** On success, a non-NULL handle will be returned. Otherwise, NULL will be returned and `GetLastError` will return the failure code.

#### 2. ReadProcessMemory (*read*)

**Prototype:**

```
BOOL ReadProcessMemory(  
    HANDLE hProcess,  
    LPCVOID lpBaseAddress,  
    LPVOID lpBuffer,  
    SIZE_T nSize,  
    SIZE_T* lpNumberOfBytesRead  
);
```

**Synopsis:** This function is exactly the same as `pread` from the procfs method in that it allows the caller to read from another process' address space into a buffer local to the calling process. This function also allows

for specifying an arbitrary buffer length, thus freeing it from any static sized limitations.

**Arguments:** The `hProcess` argument should be set to the handle that was returned from the call to `OpenProcess`. The `lpBaseAddress` argument should hold an absolute virtual memory address relative to the process being operated upon. The `lpBuffer` argument should be set to a pointer to the buffer that will hold the data that is to be read. The `nSize` argument should be set to the number of bytes that are to be read from memory. Finally, the `lpNumberOfBytesRead` argument is optional and can be set to a pointer that will hold the actual number of bytes read from the process' memory.

**Return:** On success, `TRUE` is returned. Otherwise, `FALSE` is returned and `GetLastError` will return the failure code.

### 3. `WriteProcessMemory` (*write*)

**Prototype:**

```
BOOL WriteProcessMemory(  
    HANDLE hProcess,  
    LPVOID lpBaseAddress,  
    LPCVOID lpBuffer,  
    SIZE_T nSize,  
    SIZE_T* lpNumberOfBytesWritten  
);
```

**Synopsis:** This function is exactly the same as `pwrite` from the `procfs` method in that it allows the caller to write memory from the calling process to that of another process at an arbitrary address. This function also allows for specifying an arbitrary length, thus freeing it from any static sized limitations like those found in `ptrace`.

**Arguments:** The `hProcess` argument should be set to the handle that was returned from the call to `OpenProcess`. The `lpBaseAddress` argument should hold an absolute virtual memory address relative to the process that is being operated upon. The `lpBuffer` argument should be set to a pointer to the buffer that holds the data to be written. The `nSize` argument should hold the number of bytes that are to be written. Finally, the `lpNumberOfBytesWritten` argument is optional and can be set to a pointer that will hold the actual number of bytes written to the process' memory.

**Return:** On success, `TRUE` is returned. Otherwise, `FALSE` is returned and `GetLastError` will return the failure code.

### 4. `CloseHandle` (*close*)

**Prototype:**

```
BOOL CloseHandle(  
    HANDLE hObject  
);
```

**Synopsis:** This function releases the handle that was allocated by the call to `OpenProcess` and frees up any resources that were allocated internally during the process.

**Arguments:** The `hObject` argument should be set to the handle that was returned by `OpenProcess`.

**Return:** On success, `TRUE` is returned. Otherwise, `FALSE` is returned and `GetLastError` will return the failure code.

Making use of this API is very simple and robust. The author does not know of any compatibility issues between versions of Windows with regards to the use of this API. In Windows NT-based versions one should note that proper access will be required in order to attach to a given process. For instance, regular users will not be able to attach to a process that was started by an administrator or by the system itself. This mirrors the restrictions that are set in place by the `ptrace` and `procds` methods.

## 3.2 Predicting Data Structure Layout

Before diving head long into the depths of a process' address space, one should first consider how the application could possibly be laid out. Doing this before hand can save time and reduce confusion by eliminating unneeded analysis once one actually starts looking at real data.

The first place to start this preliminary analysis is with the application's installation. What files were installed? Were shared object files installed? If so, does the executable that launches the application depend on them? This information is helpful in that it will allow the reader to identify potential places, or simply narrow it down to a single place, that are or is likely to have interesting data. The second thing to do is try to think about how the programmer of the application may have decided to implement things. Did he use a global array of structures to store things? Did he use dynamically allocated linked list nodes? What kind of information would need to be stored in the nodes? Finally, try to think of any useful static values, such as strings, that could be stored in relation to a given data structure.

These suggestions, while generic, can be made more specific given what is known about an application's purpose. Once the predication phase of memory analysis has been completed, one can then move on to the meaty portion: trying to locate and analyze data structures.

## 3.3 Locating Data Structures

This section will discuss techniques that can be applied to locate data structures in memory.

### 3.3.1 Known Value

The **Known Value** technique is used in circumstances where a value, whether it be a string, an integer, or just a set of bytes, is either known to be associated or potentially associated with a structure. The simplest example of making use of this technique can be seen when the value that is known is a string. Take for instance the following scenario:

Joe Bob the reverse engineer ninja has been tasked with analyzing the relative security of a password management suite that his work is about ready to purchase. The software itself is used to maintain a list of usernames and passwords for more than one employee in a “secure” fashion.

On the other side of the fence, SecureSoft, Inc. has written their software in the most unintelligent way. They succeed in storing the username and password list in an encrypted fashion, but they failed to remove the decrypted information from memory in a secure fashion. This leaves them open to attack from someone who has access to attach to the process while it’s running. The way they wrote the software involves decrypting the username and password list file and then translating its contents into a linked list. The linked list node is formatted as follows:

```
struct _list_item {
    char *username;
    char *password;
    struct _list_item *next;
};
```

The only thing that Joe Bob knows is his username. He forgot all his passwords since he gets to store them in this nice and “secure” password management suite. Since all he knows is his username, it seems only reasonable to try to search for that string in the address space of the password management application. Upon doing so, Joe Bob will find that, indeed, his username can be found. What good does this do him, though?

Once he has found locations in memory that hold his username string, Joe Bob can then proceed to find out what places in memory reference this string by searching for the absolute memory address of his username string. Considering the format of the `_list_item` structure, once Joe Bob finds out where his

username is being referenced, it's only a matter of time before he analyzes the structure at that address and realizes that not only is his username there, but so is his password, and all his other passwords. Needless to say, SecureSoft, Inc. is going to get fired for this one.

### 3.3.2 Contiguous Patterns

This technique is more of a conceptual process than a concrete methodology. It involves identifying patterns in contiguous memory such that one can realize when an array or a linked list is being used, for example. This is useful for trying to analyze the purpose of attributes once a structure has been found. If one sees how a field is used across multiple instances of a structure, one is more likely to be able to find patterns and determine the purposes of attributes.

The following output is an example of contiguous memory taken from the case study chapter of this document:

```
$ memgrep -p 'pidof adom' -d -a 0x082ac100 -l 820
820 bytes starting at 082ac100 (+/- 0)...
082ac100: 00000004 00000001 ffffffff 00000027
082ac110: 00ff0864 00000007 00000001 ffffffff
082ac120: 00000014 00ff0664 00000008 00000001
082ac130: ffffffff 0000002c 00ff0864 0000000e
082ac140: 00000001 ffffffff 00000002 00ff0564
082ac150: 00000010 00000001 ffffffff 00000012
082ac160: 00ff0664 00000014 00000001 ffffffff
082ac170: 00000000 00ff0233 00000015 00000001
082ac180: ffffffff 00000003 00ff0458 00000016
082ac190: 00000001 00000000 00000018 00ff0664
082ac1a0: 00000019 00000001 ffffffe 00000001
082ac1b0: 00ff0461 0000001a 00000001 ffffffff
082ac1c0: 00000004 00ff0564 0000001c 00000001
082ac1d0: ffffffff 00000000 00ff0447 00000020
082ac1e0: 00000001 ffffffff 00000000 00ff0233
082ac1f0: 00000022 00000001 ffffffff 00000000
082ac200: 00ff033c 00000024 00000001 ffffffff
...
```

It would appear from the above output that some sort of array is in use. By subtracting the distance between the common fields, one can determine that the structure in use here appears to be 80 bytes. It also seems likely that an array is in use here as the structures repeat themselves one after another with no pointers between each other as would be likely for the case of a linked list. Finally, one should be able to identify the data types of some of the fields of



the structure by the values that are set. For instance, fields with the 3 high order bytes set to zero are likely to be four byte integers, though this is not guaranteed. These are just a sample of the basic principals that can be applied when analyzing contiguous memory for patterns.

### 3.3.3 Heap Enumeration

The ability to enumerate heap allocations in a process is helpful during times when one has an address that exists inside a structure but is unable to determine the base address for the structure itself due to the lack of patterns or other indicators. If one is able to enumerate the heap one can simply walk through the list of allocations and find one that the address falls under. When a match is found not only will the base address be known, but also the size of the allocated unit, or in this case the size of the structure that was allocated.

An extension to this logic involves the ability to track where a unit of memory was allocated from. This allows the person doing the analysis the ability to bridge the gap from memory analysis to disassembly analysis such that when one finds out where a structure is being allocated from, one can then proceed to analyze what the code does with the structure. This is a more direct route to analyzing an application as one does not have to start at the entry point and work one's way through logically.

### 3.3.4 Referential Address Tree

A Referential Address Tree is a method of displaying memory address references in a more structured format. For instance, if one wants to know where the address of the string "Bob" is referenced in memory, one would simply search for the address that "Bob" occurs at. When matches are found, one may wish to know where the matched addresses are referenced at in memory, and so on. Eventually one will hit addresses that have no references in memory, assuming there are no self-referential or nested-referential addresses. The output to such a tree might look something like:

```
0x08042f0c is referenced at
  0x084700ff is not referenced
  0x08449424 is referenced at
    0x082cff94 is not referenced
  0x084cab0f is referenced at
    0x082cff98 is referenced at
      0x08cbcb0f is not referenced
```

This technique is useful for times when one is trying to understand how structures or arbitrary locations in memory are used. For instance, this technique can help with tracking down where a dynamically allocated linked list node is finally tied back to a global variable that points to the head of the list.

## 3.4 Identifying Data Structure Attribute Types

Once a data structure has been located in memory, the next task is to figure the types and purposes of some or all of the attributes in the structure's definition. This process can be made more accurate by making use of disassembly that operates on their structure's attributes as it will lend hints to the data types and offsets. Disassembly access is not required, however, though one must at least have some sort of indication as to what kind of compiler was used to generate the executable. The reason this is necessary is because compilers will pad unaligned structures differently. Finally, one should be familiar with how many bytes are associated with each kind of data type for the architecture that the application being analyzed is targeted for. This is a requirement, else one will misidentify data types.

The rest of this section will describe some of the ways one can narrow down data types of attributes. One should note that these techniques are ways to help eliminate possibilities, not ways to deterministically identify data types. The reason for this is that due to the nature of some architectures it is relatively infeasible to determine the exact type of an attribute without looking at disassembly that deals with the structure itself. In the interest of simplicity, this section will only cover these techniques as they apply to the IA-32 architecture.

### 3.4.1 Integers

There are four basic principles one should apply when trying to identify the integer data type, or `int` as it is referred to in C. These principles will help to narrow down a potential integer field. The first of the set involves the fact that integer fields should almost always be aligned on a four byte boundary. The exceptions to this rule involve structures that were instructed at compile time to not pad out to four byte boundaries. This scenario is uncommon, but not unheard of at all<sup>3</sup>. One can tell that an attribute is aligning on a four byte boundary by checking to see if the memory address is divisible evenly by four. The second principle of the set involves checking to see what the contents of the four byte value are. If the contents appear to be set to printable characters it's very likely the field is not an integer, but rather is part of or is a string of characters. This is not always the case, but it's a good item to key off of. The

---

<sup>3</sup>An example of where unaligned structures would be necessary is when one is implementing structures for the SMB protocol.

third principle one should apply is to look at the memory around the four bytes that are being looked at to get a better feel for its relation to other attributes. If the memory that surrounds the address is set to all zeroes, and so is the memory at the address, it will be very hard to conclusively say whether or not the field is actually an integer. It could be a statically sized buffer of some type, for example. Finally, one can further narrow down the type of the field by ensuring that the contents of the four bytes do not point to a valid, or potentially valid, address in memory. If the field does hold a memory address, and though the field can be construed as an integer, it is most likely a pointer of some type. With these concepts in mind, one should be able to determine whether or not the field of interest should be considered an integer or not.

Take a moment to try to determine if the following dumps should or should not be thought of as integers:

1. Address of interest: 0x081507e0

```
$ memgrep -p 'pidof adom' -d -a 081507e0 -l 16
16 bytes starting at 081507e0 (+/- 0)...
081507e0: 7373694d 20676e69 6f6c6f63 65732072
```

2. Address of interest: 0x082ac1c1

```
$ memgrep -p 'pidof adom' -d -a 0x082ac240 -l 128 -b 128
128 bytes starting at 082ac240 (+/- 128)...
082ac1c0: 00000004 00ff0564 0000001c 00000001
```

3. Address of interest: 0x082ac1c8

```
$ memgrep -p 'pidof adom' -d -a 0x082ac240 -l 128 -b 128
128 bytes starting at 082ac240 (+/- 128)...
082ac1c0: 00000004 00ff0564 0000001c 00000001
```

The first two dumps should be determined to **not likely** be integers. The reason the first dump is not likely an integer is due to the fact that each of the individual bytes are set to printable characters. Not only that, but the memory that surrounds the address is also set to printable characters. This strongly indicates this is a string, not an integer. The second dump is also not likely an integer. This is due to the fact that the address of the supposed integer is not aligned on a four byte boundary as one would expect. Finally, the last dump is likely an integer. The reason for this is that none of the four criteria used to

help to eliminate non-integer types apply and, as such, this field is likely to be a four byte integer<sup>4</sup>.

### 3.4.2 Strings

The string data type is a bit easier to identify compared to the the integer data type. The string data type can be thought of as a set of characters that are terminated by a NULL (0x00) character. One should note that when dealing with string data types, it's very likely that the string itself will not be stored inline in the structure by means of a statically sized buffer. Rather, it's more likely that the string will be referred to by a pointer. For this reason one must not only watch for whether or not attributes of a structure are set to printable characters that are terminated by a NULL character, but must also watch for attributes that are set to pointers to locations in memory that may hold a string. The first scenario is very easy to identify. All one need do is look at the memory being inspected and determine if it fits the criteria for the string data type. If so, it's a string, if not, it may or may not be. Granted, not all strings contains printable characters, but the common scenario is that they will. The latter case involving looking at pointers is slightly more complicated in that one first has to determine whether or not an attribute is set to a pointer in the first place. The process of narrowing this down involves first ensuring that the attribute is aligned on a four byte boundary as will be the case with most structures. Secondly, one should verify that the value is set to a valid address in the process' address space. Finally, one can then inspect the memory at that location to determine whether or not it fits the string data type criteria.

Take a moment to try to determine if the following dumps should or should not be thought of as strings:

1. Address of interest: 0x081507e0

```
$ memgrep -p 'pidof adom' -d -a 081507e0 -l 48
48 bytes starting at 081507e0 (+/- 0) as hexint...
081507e0: 7373694d 20676e69 6f6c6f63 65732072 Missing color se
081507f0: 6e657571 69206563 7473206e 676e6972 quence in string
08150800: 0020002e 20756f59 22007325 00227325 .. .You %s.%s".
```

2. Address of interest: 0xbfffa39

---

<sup>4</sup>One should note that this field could also be a short (2 bytes) or a char (1 byte). There is no easy way to conclusively determine this without looking at the disassembly that deals with this attribute. See the 'Other' heading later in this section for more info.

```
$ memgrep -p 'pidof adom' -d -a stack -l 32
32 bytes starting at bffffa24 (+/- 0) as hexint...
bffffa24: 4005702c 4002d829 00000000 bffffa73 ,p.@)..@....s...
bffffa34: 00000001 4002d7b6 4005702c 400554d4 .....@,p.@.T.@
```

3. Address of interest: 0xbffffa30

```
$ memgrep -p 'pidof adom' -d -a stack -l 32
32 bytes starting at bffffa24 (+/- 0) as hexint...
bffffa24: 4005702c 4002d829 00000000 bffffa73 ,p.@)..@....s...
bffffa34: 00000001 4002d7b6 4005702c 400554d4 .....@,p.@.T.@
```

The first address in the set should immediately be identifiable as a string. Each of the bytes is set to a printable character and it is terminated by a NULL character at 0x08150801. The second address in the set should not be identified as a string due to the fact that it does not align on a four byte boundary and also due to the fact none of the bytes around it are set to printable characters (except for the 0x40). Though the unaligned address is a good indicator, it is once again not conclusive with regards to eliminating string data types. Finally, the third address in the set requires a little bit more investigation. The memory address of interest seems to have a pointer stored in it. If this pointer is found to have printable characters and a NULL terminator at its location, then one should conclude that a string is being used. In this case, though, the memory at 0xbffffa73 does not hold printable characters. It is also not four byte aligned.

### 3.4.3 Pointers

Pointers are the easiest of the types to identify. Associating the word type with a pointer is a slight misnomer as pointers, by nature, are typeless. Their purpose is to point to another address in memory for use as a specific data type. That data type could be an integer, a character, or an arbitrary structure. A pointer by nature is four bytes and stores an absolute address that is relative to a process' address space. In order to identify whether or not an attribute is a pointer, one should verify that the address being pointed to is valid and that the address that holds the pointer is aligned on either a two byte or four byte boundary<sup>5</sup>.

### 3.4.4 Other

The standard data types that have yet to be identified include the **short** data type (2 bytes) and the **char** data type (1 byte). These data types are harder

<sup>5</sup>The same considerations apply with regards to packed vs unpacked structures that don't actually align on four byte boundaries.

to identify due to the fact that alignment considerations are more relaxed. For instance, short data types should almost always align on a two byte boundary, though it is not a requirement. Attributes of the char data type can align on any boundary and thus nothing can be concluded from what address they fall on. Also, it is perfectly feasible to find a structure that has two short attributes instead of an integer attribute. Take the following output for example:

```
$ memgrep -p 'pidof adom' -d -a 0x082ac240 -l 128 -b 128
128 bytes starting at 082ac240 (+/- 128)...
082ac1c0: 00000004 00ff0564 0000001c 00000001
```

Is the attribute at address 0x082ac1c0 an integer or two shorts? Or is it a short and two chars? Or is it four chars? One cannot say conclusively without looking at the disassembly that works on the structure. It is for this reason that identifying these data types is not as trivial as one might think.

## Chapter 4

# Case Study: adom

In the interest of displaying how the techniques described in this document are actually put to use, the author has taken the liberty of embarking on a quest the likes of which the world has never seen. The quest is fraught with peril for one must beware of the Great DMCA Dragon, a foul beast whom's fire laden breath is said to even scorch at the skin of children. Fear not, though, for onward does the band proceed into the realm of **Ancient Domains of Mystery** (adom). This realm is shrouded in secrecy such that the source to it is nowhere to be found. For this quest the soon-to-be mighty warrior **Reveng** will be employed.

### **To become incredibly powerful.**

In order to be introduced into the realm of adom one must first be born, and, as such, so will Reveng. Reveng is gifted from birth with excellent strategic thinking which will surely act as a boon for the quest being embarked upon. His background is as follows:

You are born in the month of the Sword on day 20 (day 200 of the year).

Aspects: Combat, Execution, Tactics, Strategy

Game effects: Positive modifiers from Tactics settings are increased by 10%, costs to increase melee weapon skills are reduced by 20%, +1 to the initial Learning score.

Background: You are born to a male gnome. You have brown hair, green eyes, and a tanned complexion. Your father is a guildmaster. Your family generally is rather wealthy and well-known in your

home town. In your childhood you were fascinated by your environment and the everchanging wonders of the world. You were a very curious and adventurous child. During your youth you had a lot of fun playing with other kids of your age. You always were a natural leader and the center of interest. As a young adult you were a credit to your family. Your parents supported your plans and were very understanding. They tried to help you along your way with all the means at their disposal. At the age of 108 you end your apprenticeship. You are now a fully learned paladin.

And so Reveng begins his quest. Will he succeed? Will he be eaten by the Great DMCA Dragon?

## 4.1 Setting Character Attribute Levels

**Techniques Used:** Known Value, Heap Enumeration

On the path to becoming the most powerful Paladin in all of adom, Reveng will first need to buff up a bit. In reality, he needs to be smarter, faster, quicker, and more wise. Being the lazy adventurer he is, though, he does not want to have to work very hard for any of this. His birth-given attributes are as follows:

```
St:14  Le:18  Wi:11  Dx:14  To:12  Ch:19  Ap:13  Ma:14  Fe:11  N=
H: 18<18>  P: 5<5>  Exp: 1/0  DrCh  Sp: 100
```

In order to change these values, one must first think about where in memory they could be located. Could they be located in the heap? Sure, adom's author could parse the save game file and store it in the heap. However, it could also be in the data section. The latter is more likely in this case given that adom is a single player game and as such does not need to store static (in the sense of size) character information in the heap.

So for starters the educated guess of the location of the attributes is the data section, but where exactly in the data section are they? And how can they be located? It's at this stage that one makes use of the **Known Value** technique. It is known that Reveng's St is 14, his Le is 18, and so on. Could it be that the attributes are stored in memory in the same order that they are presented? One can check by doing the following:

```
$ memgrep -p 'pidof adom' -s -a data \  
-f x,0e000000120000000b0000000e000000  
Searching 0x082626a0 => 0xffffffff...  
found at 0x082b1728
```



Indeed, a match was found in the data section for the attributes (as 4 byte integers). It's highly likely that this is the structure that holds character attribute information given that only one match was found and it would be very unlikely to find memory unrelated to character attribute information that had the exact same byte values. From this point one must take the address and try to see if the base address for the structure can be determined. This is useful for knowing the starting point of the character attribute structure as it may come in handy later. In order to see if the address is in the heap, one can do a quick check by enumerating the heap to see if the address falls in the range of any allocations:

```
$ memgrep -p 'pidof adom' -e -a 0x082b1728
Total matched heap entries: 0
```

Since no matches were returned, one now knows for certain that the Reveng's attributes are stored in the data section as a global variable. Determining the size of a structure that is located in the data section is not trivial given that the size of the structure is not stored anywhere other than in the text section of the binary should it reference it by `sizeof`. For this reason one should proceed with the understanding that the base address of the structure for Reveng's attributes is currently unknown. Fortunately, however, the base address is not needed to accomplish the goal of changing his attributes. The next step is to look at the memory around the address where the attributes were found. This can give more indication as to what other useful information lies around that point.

```
$ memgrep -p 'pidof adom' -d -a 0x082b1728 -b 96 -l 4
4 bytes starting at 082b1728 (+/- 96)...
082b16c8: 00000001 0000002f 00000027 00000000
082b16d8: 00000000 00000001 00000000 00000000
082b16e8: 0000004a 00000007 ffffffff ffffffff
082b16f8: 0000004a 00000007 00000000 00000000
082b1708: 00000000 00000000 00000000 00000000
082b1718: 00000006 00000007 00000003 00000005
082b1728: 0000000e 00000012 0000000b 0000000e
082b1738: 0000000c 00000013 0000000d 0000000e
082b1748: 0000000b 00000012 0000001b 00000012
082b1758: 00000017 00000014 00000017 00000011
082b1768: 00000021 00000011 00000000 00000000
082b1778: 00000000 00000000 00000000 00000001
082b1788: 00000000
```

As one can see from the above output, the **St** attribute is located at `0x082b1728` and the **Pe** attribute is located at `0x082b1748`. All the attributes in between appear in the exact same order as they do on the display screen from `adom`. With this knowledge in hand one can proceed to update all of the attribute values to the highest possible value (99).



cottages, a shop and a village hall. Trees grow peacefully between the small homes of the local populace. You feel at ease.

Sounds nice and cozy. No better environment for an unprovoked killing spree.

You hit the farmer with full force and transform him into a mangled heap! The farmer misses you.

You hit Blup, the baby water dragon and slay him. Congratulations! You advance to level 2.

Yippie! After the short killing spree Reveng managed to amass 863 experience points. This value should serve unique enough for searching purposes as it is unlikely that there are many integers in the data section set to 863.

```
$ memgrep -p 'pidof adom' -s -a data -f x,5f030000
Searching 0x082626a0 => 0xffffffff...
    found at 0x082b1700
```

After searching for 863 in the data section, only one match was found<sup>2</sup>. The key thing to note about this match is that it is only 24 bytes away from the start of the character's attributes. This is a very good indicator that the experience counter has indeed been found. Now, in order to accomplish this phase of the quest, Reveng must be increased to level 50<sup>3</sup>. The requirement for level 50 is 18651974 experience points. So, given that, one should proceed by updating the experience counter location in memory to 18651974.

```
$ memgrep -p 'pidof adom' -r -a 0x082b1700 -t i,18651974
Replacing memory 082b1700 with 4 bytes of data...
```

After updating the experience counter, all Reveng had to do after that point was move one more step down the road so that adom's internal state machine could recognize that an experience threshold had been reached. Indeed, it pained him much to walk one step...

H:1178<1291> P:1670<1670> Exp: 50/18651974

The script that implements this design can be found in the appendix (A.2).

---

<sup>2</sup>The data section is used due to the fact that the character attributes were located in the data section.

<sup>3</sup>adom lists the experience count requirements for each level inside the game.

## 4.3 Setting Character Skill Levels

**Techniques Used:** Known Value, Contiguous Patterns

At this stage Reveng is a buff and powerful little paladin, but sad to say he's got a limited skill set. If he hopes to get a job as a Master Dragon Slayer he'll need to spruce up his skills a bit before applying. The only way he can hope to become Master Dragon Slayer is to make it so his character has the maximum of every skill possible, even skills he wouldn't otherwise normally have access to! His starting skills are as follows:

A	- Athletics	100	<superb>	[+2d4]
B	- Climbing	100	<superb>	[+1d5]
C	- Concentration	100	<superb>	[+2d4]
D	- Dodge	100	<superb>	[+1d3]
E	- First aid	100	<superb>	[+1d5]
F	- Genealogy	51	<good>	[+1]
G	- Hagglng	88	<great>	[+1d3]
H	- Healing	100	<superb>	[+1d5]
I	- Listening	97	<superb>	[+1d3]
J	- Literacy	100	<superb>	[+1d3]
K	- Mining	71	<great>	[+1]
L	- Pick pockets	51	<good>	[+1]
M	- Stealth	60	<good>	[+1]
N	- Swimming	85	<great>	[+1d3]
O	- Two weapon combat	100	<superb>	[+1d3]
P	- Ventriloquism	61	<good>	[+1]

Given the above list one knows the values associated with various skills, but unlike character attributes, skills are dynamically associated with a character and as such may potentially be stored in the heap, vice being stored in the data section. Also, one knows the names of the currently obtained skills. This can be useful for trying to back track things that reference the strings in the data and bss sections. It's the latter approach that will be focused on initially.

The skill that will be worked back from will be the "Two Weapon Combat" skill. The first thing to do is to search rodata for the string "Two Weapon Combat".

```
$ memgrep -p 'pidof adom' -s -a rodata -f "s,Two weapon combat"
Searching 0x081507a0 => 0xffffffff...
    found at 0x081cc35c
    found at 0x082047e8
    found at 0x082cc463
```

This search doesn't quite narrow it down enough, so perhaps if one searches for another skill name they might find it to be close in memory to the "Two weapon combat" entry.

```
$ memgrep -p 'pidof adom' -s -a rodata -f "s,Ventriloquism"
Searching 0x081507a0 => 0xffffffff...
    found at 0x081cc34e
    found at 0x081f5b6b
    found at 0x081ffdb2
    found at 0x08204608
    found at 0x082cc504
```

Indeed, it appears that both strings are relatively close to each other in memory in three different places. What does the memory at these locations look like?

```
$ memgrep -p 'pidof adom' -d -a 0x081cc34e -l 64 -c
64 bytes starting at 081cc34e (+/- 0)...
081cc34e: Ventriloquism.Tw
081cc35e: o weapon combat.
081cc36e: Tactics.Swimming
081cc37e: .Survival.Stealt
```

```
$ memgrep -p 'pidof adom' -d -a 0x08204608 -l 64 -c
64 bytes starting at 08204608 (+/- 0)...
08204608: Ventriloquism. T
08204618: his skill can be
08204628: used to confuse
08204638: monsters..skill
```

```
$ memgrep -p 'pidof adom' -d -a 0x082cc504 -l 64 -c
64 bytes starting at 082cc504 (+/- 0)...
082cc504: Ventriloquism ..
082cc514: ..... ..0m...37
082cc524: m..40m 61..m...3
082cc534: 9.49m..37m..40m
```

It would appear that any of those addresses could be useful, but the top most one appears most interesting as it is the list of all the skill names by themselves. For that reason one can proceed to search the data and bss sections for places in memory that have a pointer to the string 'Ventriloquism'.

```
$ memgrep -p 'pidof adom' -s -a data,bss -f x,4ec31c08
Searching 0x082626a0 => 0xffffffff...
    found at 0x082884b8
Searching 0x0829e5c0 => 0xffffffff...
```

Ah-ha! A match! So what does the memory look like at this location?

```
$ memgrep -p 'pidof adom' -d -a 0x082884b8 -l 96
96 bytes starting at 082884b8 (+/- 0)...
082884b8: 081cc34e 34333333 38343434 34343434
082884c8: 32343234 35333433 00000000 00000000
082884d8: 00000000 00000005 081cc344 33383334
082884e8: 36333333 35323233 37323634 34333236
082884f8: 00000000 00000000 00000001 00000003
08288508: 00000011 00000000 00000000 00000000
```

Interesting, that looks like a pattern. What does 0x081cc344 point to?

```
$ memgrep -p 'pidof adom' -d -a 0x081cc344 -l 64 -c
64 bytes starting at 081cc344 (+/- 0)...
081cc344: Woodcraft.Ventri
081cc354: loquism.Two weap
081cc364: on combat.Tactic
081cc374: s.Swimming.Survi
```

Very interesting. So it would appear that the memory at 0x082884b8 is actually an element in an array of structures. This element is for the **Ventriloquism** skill. The pattern can be seen by looking 0x28 bytes after the Ventriloquism skill element. The same scenario seems to repeat itself, a pointer to the name of the skill, 20 bytes of numbers (as ascii), and then 16 bytes of unknown integer values. The interesting difference between the two elements is that the last two integers are different. This could be handy later.

At this point an array that is associated with skills has been identified. In order to go further one will want to find the top of the array, at which point more analysis can be done. The process of finding the top of the array involves dumping memory around that address (continually decreasing) until the pattern stops repeating itself. This is a good indication that the top of the array has been hit.

Following this logic, the top of the array can be determined to be 0x08287ea0. Doing some quick math, one can determine exactly how many skills are in the game. The expression is:

$$(\text{endaddr} - \text{beginaddr}) / \text{elementsized} = \text{numskills}$$

In this case, endaddr is 0x08288508, or the Woodcraft skill. The beginaddr is 0x08287ea0 as described above. The element size is 40 bytes as can be calculated by the the number of bytes between the two pointers for the names of the skills. By substituting these values into the expression, one comes to the conclusion that there are exactly 41 skills in the game.

Very good then, progress is being made, Reveng will soon have all the skills he ever dreamed of. The hard part still remains, though. Where exactly are the skills that associated with the character? So far an array has been found that houses all of the skills, but is it also used to store whether or not the a skill is associated with a character? And, if so, does it store the current level of the skill associated with the character? If one looks at the differences between the elements in the array, one will notice that they do not appear to house the skill level information for the current character. This is made clear by the fact that for Ventriloquism's entry there is no byte in the element that is set to

'61'. If this array doesn't house the skill information, it's time to begin looking elsewhere with this tidbit of information safely locked away.

The first thing to rule out is whether or not this array is referenced elsewhere in memory. If it is then the place that references it may be of great value. However, as one can see, there is no such reference in memory.

```
$ memgrep -p 'pidof adom' -s -a data,bss -f x,a07e2808
Searching 0x082626a0 => 0xffffffff...
Searching 0x0829e5c0 => 0xffffffff...
```

Drats. No matches. Looks like this might not be as easy as was initially thought. Time to put the thinking cap on and think like a programmer. If one were to write code to manage a static set of things (in this case skills), one would probably make use of either a linked list or an array. Considering the fact that the number of elements is indeed static, one would think it might be more likely that an array would be used. This is also emphasized by the fact that adom's programmers used an array to store other information about skills in another context. Given these two points of evidence, what other useful values could be used to search for? One of the more obvious ones would be to search for the value associated with one or more skills. Unfortunately, however, this will return no results that are of any relevance. For example, if one searches for the integer '100' in the bss, one will find matches that are not near each other or have no discernable pattern.

One technique that can be employed in a situation such as this one is to dump the entire contents of bss and walk through the output until patterns are noticed. This technique can be used to solve this problem, but in the interest of education it will not be used. Instead, one can use a slightly more creative approach. Ponder the following: It is known that there are 41 skills in the game. Given that, it is also known that 42 would be one past the number of skills that are in the game. Taking a shot in the dark, one might do the search bss for the integer '42'. Doing this, one finds that the rewards are great:

```
$ memgrep -p 'pidof adom' -s -a bss -f i,42 Searching 0x0829e5c0
=> 0xffffffff...
...
foundn at 0x082ac240
found at 0x082ac254
found at 0x082ac268
found at 0x082ac27c
found at 0x082ac290
found at 0x082ac2a4
found at 0x082ac2b8
found at 0x082ac2cc
```

```
found at 0x082ac2e0
found at 0x082ac2f4
found at 0x082ac308
found at 0x082ac31c
found at 0x082ac330
found at 0x082ac344
found at 0x082ac358
found at 0x082ac36c
found at 0x082ac380
found at 0x082ac394
...
```

The above output is abbreviated in the interest of space, but as one can see there is a clear pattern in the above search. This may help bolster the theory that arrays are being used to store skills if this does indeed have anything to do with skills. In order to determine if the matches are at all applicable, one should dump the memory at the first match to fit the pattern and see what surrounds it.

```
$ memgrep -p 'pidof adom' -d -a 0x082ac240 -l 128 -b 128
128 bytes starting at 082ac240 (+/- 128)...
082ac1c0: 00000004 00ff0564 0000001c 00000001
082ac1d0: ffffffff 00000000 00ff0447 00000020
082ac1e0: 00000001 ffffffff 00000000 00ff0233
082ac1f0: 00000022 00000001 ffffffff 00000000
082ac200: 00ff033c 00000024 00000001 ffffffff
082ac210: 00000002 00ff0455 00000026 00000001
082ac220: ffffffff 00000002 00ff0564 00000027
082ac230: 00000001 ffffffff 00000000 00ff033d
082ac240: 0000002a 00000000 00000000 00000000
082ac250: 00ffffff 0000002a 00000000 00000000
082ac260: 00000000 00ffffff 0000002a 00000000
082ac270: 00000000 00000000 00ffffff 0000002a
082ac280: 00000000 00000000 00000000 00ffffff
082ac290: 0000002a 00000000 00000000 00000000
082ac2a0: 00ffffff 0000002a 00000000 00000000
082ac2b0: 00000000 00ffffff 0000002a 00000000
082ac2c0: 00000000 00000000 00ffffff 0000002a
082ac2d0: 00000000 00000000 00000000 00ffffff
082ac2e0: 0000002a 00000000 00000000 00000000
082ac2f0: 00ffffff 0000002a 00000000 00000000
082ac300: 00000000 00ffffff 0000002a 00000000
082ac310: 00000000 00000000 00ffffff 0000002a
082ac320: 00000000 00000000 00000000 00ffffff
082ac330: 0000002a 00000000 00000000 00000000
```



Well, a pattern has indeed been identified. In order to determine if what is in the array is at applicable one could start by finding the start point, end point, and element size of the array. With these numbers a determination can be made as to whether or not this is potentially related to skills. If it is related to skills the number of elements should be exactly or near 41.

Given the memory dump above one can determine that the size of the elements in the array is 20 bytes given that the elements that match the pattern have their 0x00ffffff fields 20 bytes apart. Now that the element size is known, it would potentially be easiest to find the beginning point of the array as elements nearest the end don't have enough information to determine where exactly the start of each element is. The beginning point can be determined by moving down in memory until the trend begins to change. Once that happens, one can then try to determine where the first element begins, and, in so doing, identify the start of the array.

```
$ memgrep -p 'pidof adom' -d -a 0x082ac0e0 -l 96
96 bytes starting at 082ac0e0 (+/- 0)...
082ac0e0: 00000000 00000000 00000000 00000000
082ac0f0: 00000000 00000000 00000000 00000000
082ac100: 00000004 00000001 ffffffff 00000027
082ac110: 00ff0864 00000007 00000001 ffffffff
082ac120: 00000014 00ff0664 00000008 00000001
082ac130: ffffffff 0000002c 00ff0864 0000000e
```

As can be seen from the above memory dump, the pattern begins to change somewhere above 0x082ac100. Interestingly enough, though, is that one can notice that 0x082ac100 is also the starting point for the array by noticing that that, given an element size of 20 bytes as determined above, the next element would start at 0x082ac114. This element also follows the exact same pattern as the first one. The final step now is to walk up into memory 20 bytes at a time to figure out where the pattern stops.

```
$ memgrep -p 'pidof adom' -d -a 0x082ac400 -l 96
96 bytes starting at 082ac400 (+/- 0)...
082ac400: 00000000 00000000 00ffffff 0000002a
082ac410: 00000000 00000000 00000000 00ffffff
082ac420: 0000002a 00000000 00000000 00000000
082ac430: 00ffffff 00000000 00000000 00000000
082ac440: 00000000 00000000 082d6540 00000000
082ac450: 00000000 00000000 00000000 00000000
```

Since the pattern for the element structure is known, one can determine the last element to ends at 0x082ac434. If one takes the determined endaddr

(0x082ac434), beginaddr (0x082ac100), and element size (20) as determined through the process, one will find that this array holds **41** elements, exactly matching the number of skills that were found earlier. This is great news! Now to determine if this is actually related to the current characters skill levels. The easiest way to approach this is to dump the entire array and analyze it. Given that there are 41 elements each 20 bytes in size, that would mean that the array takes up a total of **820** bytes.

```
$ memgrep -p 'pidof adom' -d -a 0x082ac100 -l 820
820 bytes starting at 082ac100 (+/- 0)...
082ac100: 00000004 00000001 ffffffff 00000027
082ac110: 00ff0864 00000007 00000001 ffffffff
082ac120: 00000014 00ff0664 00000008 00000001
082ac130: ffffffff 0000002c 00ff0864 0000000e
082ac140: 00000001 ffffffff 00000002 00ff0564
082ac150: 00000010 00000001 ffffffff 00000012
082ac160: 00ff0664 00000014 00000001 ffffffff
082ac170: 00000000 00ff0233 00000015 00000001
082ac180: ffffffff 00000003 00ff0458 00000016
082ac190: 00000001 00000000 00000018 00ff0664
082ac1a0: 00000019 00000001 fffffffe 00000001
082ac1b0: 00ff0461 0000001a 00000001 ffffffff
082ac1c0: 00000004 00ff0564 0000001c 00000001
082ac1d0: ffffffff 00000000 00ff0447 00000020
082ac1e0: 00000001 ffffffff 00000000 00ff0233
082ac1f0: 00000022 00000001 ffffffff 00000000
082ac200: 00ff033c 00000024 00000001 ffffffff
082ac210: 00000002 00ff0455 00000026 00000001
082ac220: ffffffff 00000002 00ff0564 00000027
082ac230: 00000001 ffffffff 00000000 00ff033d
082ac240: 0000002a 00000000 00000000 00000000
082ac250: 00ffffff 0000002a 00000000 00000000
082ac260: 00000000 00ffffff 0000002a 00000000
082ac270: 00000000 00000000 00ffffff 0000002a
082ac280: 00000000 00000000 00000000 00ffffff
082ac290: 0000002a 00000000 00000000 00000000
082ac2a0: 00ffffff 0000002a 00000000 00000000
082ac2b0: 00000000 00ffffff 0000002a 00000000
082ac2c0: 00000000 00000000 00ffffff 0000002a
082ac2d0: 00000000 00000000 00000000 00ffffff
082ac2e0: 0000002a 00000000 00000000 00000000
082ac2f0: 00ffffff 0000002a 00000000 00000000
082ac300: 00000000 00ffffff 0000002a 00000000
082ac310: 00000000 00000000 00ffffff 0000002a
082ac320: 00000000 00000000 00000000 00ffffff
```

```

082ac330: 0000002a 00000000 00000000 00000000
082ac340: 00ffffff 0000002a 00000000 00000000
082ac350: 00000000 00ffffff 0000002a 00000000
082ac360: 00000000 00000000 00ffffff 0000002a
082ac370: 00000000 00000000 00000000 00ffffff
082ac380: 0000002a 00000000 00000000 00000000
082ac390: 00ffffff 0000002a 00000000 00000000
082ac3a0: 00000000 00ffffff 0000002a 00000000
082ac3b0: 00000000 00000000 00ffffff 0000002a
082ac3c0: 00000000 00000000 00000000 00ffffff
082ac3d0: 0000002a 00000000 00000000 00000000
082ac3e0: 00ffffff 0000002a 00000000 00000000
082ac3f0: 00000000 00ffffff 0000002a 00000000
082ac400: 00000000 00000000 00ffffff 0000002a
082ac410: 00000000 00000000 00000000 00ffffff
082ac420: 0000002a 00000000 00000000 00000000
082ac430: 00ffffff

```

The first thing to do is to try to determine the structure associated with each array element. At a high-level it would appear that each element has the first 4-byte integer set to a number that is set to something less than or equal to 42. Also, none of the integers repeat themselves except for the case of the number 42. This could lend credence to the fact that the first 4-byte integer is an identifier for a skill. The next 4-byte integer is set to 1 for every element that has the first 4-byte integer set to something other than 42. If one assumes that the first 4-byte integer is an identifier and that it's known that 42 is outside of the range of game skills, it should be safe to assume that the second 4-byte integer is an enabled/disabled field. It's set to 1 if the skill is enabled, otherwise 0 if it's disabled. The next three 4-byte fields are a little bit more challenging to identify.

The last 4-byte field is the easiest one to start with. At this point it is useful to remember what order the skills are displayed in on the user interface. According to the display, the first 5 skills are maxed out at level 100. If one looks at the first byte of the last 4-byte field for the first 5 elements one will notice that each of them is set to 0x64, or 100 in decimal. To validate the pattern, the 6th skill, Gemology, is set at level 51. If one goes to the first byte of the last 4-byte field for the 6th element, one will indeed verify that it is set to 0x33, or 51 in decimal. Finally! The skills array associated with Reveng has been located! The determined structure can be written in C-style as follows:

```

struct _character
{
    unsigned long skillId; // offset 0x00
    unsigned long enabled; // offset 0x04

```

```

    unsigned long unknown1; // offset 0x08
    unsigned long unknown2; // offset 0x0c
    unsigned char level;    // offset 0x10
    unsigned char unknown3; // offset 0x11
    unsigned short tail;    // offset 0x12 (always 0x00ff)
};

```

Now that the pertinent fields have been identified, perhaps it's time to take a test run on Reveng's Gemology skill. No one is anyone without being level 100 in their Gemology skill. As was determined before, the Gemology skill is the 6th element in the array, or address 0x082ac164. The skill level is 16 bytes into that element, or 0x082ac174.

```

$ memgrep -p 'pidof adom' -r -a 082ac174 -t x,64
Replacing memory 082ac174 with 1 bytes of data...

```

And with that, Reveng now has level 100 in his Gemology skill.

```

A - Athletics ..... 100 (superb) [+2d4]
B - Climbing ..... 100 (superb) [+1d5]
C - Concentration ..... 100 (superb) [+2d4]
D - Dodge ..... 100 (superb) [+1d3]
E - First aid ..... 100 (superb) [+1d5]
F - Gemology ..... 100 (superb) [+1]
G - Hagglng ..... 88 (great) [+1d3]
H - Healing ..... 100 (superb) [+1d5]
I - Listening ..... 97 (superb) [+1d3]
J - Literacy ..... 100 (superb) [+1d3]
K - Mining ..... 71 (great) [+1]
L - Pick pockets ..... 51 (good) [+1]
M - Stealth ..... 60 (good) [+1]
N - Swimming ..... 85 (great) [+1d3]
O - Two weapon combat ..... 100 (superb) [+1d3]
P - Ventriloquism ..... 61 (good) [+1]

```

With most of the hard work behind, all that's left to accomplish is to figure out how to give Reveng skills that he doesn't even have. The first approach would be to change an element in the array that has been assigned id 42 to a valid identifier and other valid settings. To test this theory, one should pick an identifier that hasn't already been used, like identifier 0x17, or 23 in decimal. Also, in order to make this element like the other ones, one must set the enabled attribute to 1. The first unused skill element can be found at 0x082ac240.

```

$ memgrep -p 'pidof adom' -r -a 082ac240 -t x,1700000001000000
Replacing memory 082ac240 with 8 bytes of data...

```

So, does Reveng have a new skill now? Indeed, he does.

```

A - Athletics ..... 100 (superb) [+2d4]
B - Climbing ..... 100 (superb) [+1d5]
C - Concentration ..... 100 (superb) [+2d4]
D - Dodge ..... 100 (superb) [+1d3]
E - First aid ..... 100 (superb) [+1d5]
F - Gemology ..... 100 (superb) [+1]
G - Hagglng ..... 88 (great) [+1d3]
H - Healing ..... 100 (superb) [+1d5]
I - Listening ..... 97 (superb) [+1d3]
J - Literacy ..... 100 (superb) [+1d3]
K - Mining ..... 71 (great) [+1]
L - Pick pockets ..... 51 (good) [+1]
M - Stealth ..... 60 (good) [+1]
N - Swimming ..... 85 (great) [+1d3]
O - Two weapon combat ..... 100 (superb) [+1d3]
P - Ventriloquism ..... 61 (good) [+1]
Q - Herbalism ..... -1 (poor) [+1d5]

```

With that last bit of information it has been successfully verified that one can manipulate existing skills, as well as add new skills, to any given character. The process of identifying what skill id's are what could be approached by simply adding a new skill with a different id each time to see what it's displayed as, for instance. Now that all of this knowledge has been obtained, one could also write a simple script to go through and add every skill to a character. After running such a script, a portion of Reveng's skill display looks like this<sup>4</sup>:

```
A - Alertness ..... 100 (superb) [+1d5]
B - Appraising ..... 100 (superb) [+1]
C - Archery ..... 100 (superb) [+1d3]
D - Athletics ..... 100 (superb) [+2d4]
E - Backstabbing ..... 100 (superb) [+1d5]
F - Bridge building ..... 100 (superb) [+1d3] (mr)
G - Climbing ..... 100 (superb) [+1d5]
H - Concentration ..... 100 (superb) [+2d4]
I - Cooking ..... 100 (superb) [+1] (mr)
J - Courage ..... 100 (superb) [+1d5]
K - Detect item status ..... 100 (superb) [+1d5]
L - Detect traps ..... 100 (superb) [+1d3]
M - Disarm traps ..... 100 (superb) [+1d5]
N - Dodge ..... 100 (superb) [+1d3]
O - Find weakness ..... 100 (superb) [+1d3]
P - First aid ..... 100 (superb) [+1d5]
Q - Fletchery ..... 100 (superb) [+1] (mr)
R - Food preservation ..... 100 (superb) [+1d3]
S - Gardening ..... 100 (superb) [+1]
T - Gemology ..... 100 (superb) [+1]
U - Hagglng ..... 100 (superb) [+1d3]
V - Healing ..... 100 (superb) [+1d5]
W - Herbalism ..... 100 (superb) [+1d5]
X - Law ..... 100 (superb) [+1d5]
Y - Listening ..... 100 (superb) [+1d3]
```

Phew. That was a long one, but boy is Reveng satisfied. Well, at least somewhat satisfied. He's still quite miffed that he's made it to level 50, has all of his skills and attributes maxed out, but to date all he has on his personage are 65 gold pieces and a few tidbits of clothes. This just wont do. And with that, the quest continues...

The script that implements this design can be found in the appendix (A.3).

## 4.4 Changing Item Attributes

**Techniques Used:** Known Value, Contiguous Patterns, Heap Enumeration

With the wind blowing softly from the east, Reveng finds himself sitting quietly, sobbing himself a new river because he doesn't have the latest Chain Mail Helmet. Fear not little warrior!

For starters, here are the list of equipped and unequipped items that Reveng entered the realm with:

```
leather cap [+0, +0] [5s]
chain mail [-1, -3] [-3, +5] [400s]
-
short sword [+0, 1d6] [30s]
small shield [+2, +0] [50s]
-
-
gnomish boots [+1, +0] [+2, +0] [10s]
-
-
```

<sup>4</sup>The script referenced in this section can be found in the appendix.

```

Tools <P>
A - heap of 4 torches [40s]
B - box with flint and steel <41> [5s]
C - tinderbox <9> [3s]
D - holy symbol [5s]
Potions <P>
E - oil of rust removal [2s]
Food <P>
F - iron ration [100s]
Valuable <P>
G - 65 gold pieces [0s]

```

The simplest point to start at for this leg of the quest would be at the search for one of the items Reveng currently has on him. A quick check of his inventory reveals a set of gnomish boots! That sounds like as good of a starting point as any. The approach taken here is much the same as the previous section: start with a known string, in this case 'gnomish boots', and try to work backwards to see if any interesting places in memory reference that string. Considering that the set of items a character has on them is dynamic, it would seem fair to say that a character's inventory is stored in a linked list. As such, it's very likely that item structures will be found in bss.

```

$ memgrep -p 'pidof adom' -s -a rodata -f s,"gnomish boots"
Searching 0x081507a0 => 0xffffffff...
  found at 0x081b635a
  found at 0x081b6368
  found at 0x081fffe5
  found at 0x082a3580
  found at 0x082a84c4
  found at 0x082cbf67

```

Six matches leaves much to be desired, but alas, some investigation is required. What kind of data surrounds these strings? That may help with narrowing down the one that should be used.

```

$ memgrep -p 'pidof adom' -d -a 0x081b6368 -l 64 -c -b 32
64 bytes starting at 081b6368 (+/- 32)...
081b6348: arhammer.pairs o
081b6358: f gnomish boots.
081b6368: gnomish boots.pa
...

```

```

$ memgrep -p 'pidof adom' -d -a 0x081b635a -l 64 -c -b 32
64 bytes starting at 081b635a (+/- 32)...
...
081b634a: hammer.pairs of
081b635a: gnomish boots.gn
081b636a: omish boots.pair
...

```

```

$ memgrep -p 'pidof adom' -d -a 0x081fffe5 -l 64 -c -b 32

```

```

64 bytes starting at 081fffe5 (+/- 32)...
...
081fffd5: her workers and
081fffe5: gnomish boots ar
081ffff5: e a quality.....
...

$ memgrep -p 'pidof adom' -d -a 0x082a3580 -l 64 -c -b 32
64 bytes starting at 082a3580 (+/- 32)...
...
082a3570: .....
082a3580: gnomish boots ..
082a3590: 1. .0. ..2. .0..
...

$ memgrep -p 'pidof adom' -d -a 0x082a84c4 -l 64 -c -b 32
64 bytes starting at 082a84c4 (+/- 32)...
...
082a84b4: .....
082a84c4: gnomish boots.r
082a84d4: 702.data.....
...

$ memgrep -p 'pidof adom' -d -a 0x082cbf67 -l 64 -c -b 32
64 bytes starting at 082cbf67 (+/- 32)...
...
082cbf57: ...37m..40m
082cbf67: gnomish boots ..
082cbf77: 1. .0. ..2. .0..
082cbf87: .74G.10s....17d.
...

```

The first address in the group seems to be the most useful. The reason for this is that it is the only singular (as in not inside another string) null-terminated string that exists in the rodata section. The data section begins at 0x082626a0 and thus the last three matches can be eliminated for now. The reason they can be eliminated is because at this point the thing that is trying to be found is a structure, either in the data section or bss, that references the the string "gnomish boots". Now that the string has been found, the next step is to try to look in both the data and bss sections to see if the pointer to the string is referenced. If so, it's likely that the thing referencing it is a structure that has to do with the "gnomish boots" item.

```

$ memgrep -p 'pidof adom' -s -a data,bss -f x,68631b08
Searching 0x082626a0 => 0xffffffff...

```

```
found at 0x08274cfc
Searching 0x0829e5c0 => 0xffffffff...
```

Excellent. One match, and it's in the data section. This should be setting alarms off in the reader's head (deja-vu?). It would appear that history is repeating itself in the form of an array of structures located in the data section, very much similar to how non-character related information about skills was stored. The proof, however is still lacking. The easiest way to find out if this fits the same pattern is to dump the memory at this location and check for any obvious patterns.

```
$ memgrep -p 'pidof adom' -d -a 0x08274cfc -l 256
256 bytes starting at 08274cfc (+/- 0)...
08274cfc: 081b6368 081b6351 081b6757 081b6757
08274d0c: 00000002 00000000 00000001 00000000
08274d1c: 00000001 00000003 00000000 00000000
08274d2c: 00000001 00000003 0000000a 0000000c
08274d3c: 00000000 00000001 00000306 00000000
08274d4c: 00000000 00000032 00000003 ffffffff
08274d5c: 00000000 00000009 00000000 ffffffff
08274d6c: 081b6347 081b633c 081b6757 081b6757
08274d7c: 00000000 00000000 00000000 00000000
08274d8c: 00000002 00000006 00000000 00000000
08274d9c: 00000001 00000004 0000003c 00000006
08274dac: 00000000 00000001 00000340 00001000
08274dbc: 00000000 000001f4 00000000 00000002
08274dcc: 00000000 00000001 00000000 ffffffff
08274ddc: 081b6326 081b630f 081b6757 081b6757
```

Yeeeeees, Smithers, it would appear a pattern has been found. From the looks of it an array has been located. Just like last time, the first thing to do is locate top of the array by walking down in memory until the pattern stops repeating itself. In this case the pattern appears to stop at 0x08273afc. One can notice the pattern diverge at this point by looking at the following dump:

```
$ memgrep -p 'pidof adom' -d -a 0x08273acc -l 256
256 bytes starting at 08273acc (+/- 0)...
08273acc: 081ae472 081ae469 081ae463 081ae457
08273adc: 081ae451 081ae44b 081ae443 081ae43b
08273aec: 081ae435 081ae430 081ae426 081ae421
08273afc: 00000000 00000009 00000000 ffffffff
08273b0c: 081b6760 081b6758 081b6757 081b6757
08273b1c: 00000000 00000000 00000000 00000000
08273b2c: 00000001 00000004 00000000 00000000
```



```

08273b3c: 00000001 00000004 0000000a 00000002
08273b4c: 00000000 00000001 00000340 00001000
08273b5c: 00000000 000007d0 00000000 00000001
08273b6c: 00000000 00000009 00000000 ffffffff
08273b7c: 081b6752 081b674c 081b6757 081b6757
08273b8c: 00000000 00000000 00000000 00000000
08273b9c: 00000001 00000006 00000000 00000000
08273bac: 00000001 00000004 0000001e 00000001
08273bbc: 00000000 00000001 00000340 00001000

```

With the start point identified, one can proceed to determine the size of the structure by looking at the point of divergence. In this case, one notes that the structure begins to complete it's cycle again starting from the point 0x08273afc and repeating again at 0x08273b6c. A quick subtraction shows that the size of the elements in this array is **112** bytes. With that, the last item left to determine is the end point of the array, at which point one can calculate the number of elements. After walking up memory addresses, the end point can finally be found to be at 0x08286bec. Another quick calculation and one finds that the number of elements in the array is (drum roll): **697**.

The array of game items has been found and identified, but what good does it do? Reveng is still sobbing over not having the latest and greatest Gnomish Pumps™. The approach taken here is not a deterministic one. A person could choose to try to take apart the item structure for the individual elements found in the array, or, instead of trying to analyze the element structure, one could try to find references to individual elements in memory. These references are most likely to be found in the bss due to the consideration listed above regarding how characters have a dynamic list of items. From the calculations above, the “gnomish boots” structure actually starts at 0x08274cec, not 0x08274cfc as was found when searching for references to the string “gnomish boots”. With this address one can search bss (and the data section, potentially), for references to this item in the array. Dynamic items are likely to be stored in one of two ways: Either as a pointer to the element in the array, or, as an index into the array. The first thing to check is the pointer, as it is likely to has less false positives.

```

$ memgrep -p 'pidof adom' -s -a data,bss -f x,ec4c2708
Searching 0x082626a0 => 0xffffffff...
Searching 0x0829e5c0 => 0xffffffff...

```

Drats, no matches for the pointer to the element. The element index for Reveng's “gnomish boots” can be calculated by doing:

$$(\text{elementaddr} - \text{startaddr}) / \text{elementsize} = \text{element index}$$

Doing the above calculation yields an element index of **41**. Perhaps a reference

to the integer 41 is stored somewhere in memory? The problem with searching for the integer 41 in memory is that it's likely to get a bunch of matches. This will make the determination process much harder, but not impossible.

```
$ memgrep -p 'pidof adom' -s -a bss -f i,41
Searching 0x0829e5c0 => 0xffffffff...
...
found at 0x082e22a8
found at 0x082e2e08
found at 0x082f7170
found at 0x082f7370
found at 0x082f93e8
found at 0x082f977c
found at 0x082fae0c
...
```

In the interest of space, only the “interesting” matches are shown above. In total there were about 25 matches. Many of them can be quickly discarded to the fact that no obvious pattern can be recognized. There are some, however, that are less obvious and require a closer look. There are a few things to keep in mind when analyzing data structures in the heap. First of all, it is likely that elements in a linked list will have items located near each other in memory<sup>5</sup>. Secondly, data structures should have a similar pattern; in the case of a list, look for previous and next pointers in the same locations, as well as other values in the same locations relative to one another. Keep these concepts in mind during the following steps.

```
$ memgrep -p 'pidof adom' -d -a 0x082e22a8 -l 256
256 bytes starting at 082e22a8 (+/- 0)...
082e22a8: 00000029 00000000 00000000 00000011
082e22b8: 082e22a8 00000002 082e2298 00000019
082e22c8: 0000005a 082e22f0 00000001 082e2280
082e22d8: 00000000 00000011 082e22c8 00000010
082e22e8: 082e22b8 00000011 0000005b 00000000
082e22f8: 00000000 00000011 082e22f0 00000002
082e2308: 082e22e0 00000019 00000020 082e2338
...
```

**0x082e22a8:** This address does seem to be in the middle of a linked list of some sorts. One can tell that it may be a linked list by noting that the addresses stored in memory seem to reference one another, or at least make references near one another. While not conclusive, this is always a good indicator. Also, the pattern seems to indicate that some other item indexes may be in use. This

---

<sup>5</sup>This is dependent on the implementation of the user-mode memory manager.

can be validated by converting the index into an absolute memory address and determining if Reveng actually has the item being referred to on his personage. For instance, take the identifier **0x5b** from above. To equation used to take an index back to an absolute address is:

$$(\text{index} * \text{elementsized}) + \text{baseaddr} = \text{element address}$$

After doing the above calculation, one can determine that the item referred to by identifier 0x5b, or 91, is **0x082762cc**. In previous steps it was determined that 0x10 bytes into this structure is a pointer to the name of the item. When one this pointer, one sees that the item being referenced is named a “holy symbol”. This is interesting because Reveng does indeed have a “holy symbol” in his inventory. This could very well be the item list, but it seems to convey too little information. In order to get more information on this, one can make use of memgrep’s heap enumeration feature to see what the base address for the allocated unit is, as well as its size.

```
$ memgrep -p 'pidof adom' -e -a 0x082e22a8
Found allocation unit for 082e22a8: base is 082e22a8, length is 2.
Total matched heap entries: 1
```

Interesting, this unit seems to only be two bytes in size. That would make it the character ‘A’, plus a NULL terminator. Something smells fishy here. At this point it would make sense to flag this address as “interesting”, but this new tidbit seems to indicate that it’s unrelated and that the “holy symbol” match was just a coincidence.

```
$ memgrep -p 'pidof adom' -d -a 0x082e2e08 -l 256
256 bytes starting at 082e2e08 (+/- 0)...
082e2e08: 00000029 082e2e30 00000001 082e2dc0
082e2e18: 00000000 00000011 082e2e08 00000010
082e2e28: 082e2df8 00000011 00000069 00000000
082e2e38: 00000000 00000011 082e2e30 00000002
082e2e48: 082e2e20 00000019 00000065 082e2e78
082e2e58: 00000003 082e2e08 00000000 00000011
082e2e68: 082e2e50 00000010 082e2e40 00000011
082e2e78: 0069435c 00000000 00000000 00000011
082e2e88: 082e2e78 00000004 082e2e68 00000019
082e2e98: 00000061 082e2ec0 00000003 082e2e50
```

**0x082e2e08**: This linked list just looks like the same format as the previously mentioned address. As such, it is not likely that this address is relative to the task at hand.

```

$ memgrep -p 'pidof adom' -d -a 0x082f7170 -l 256
256 bytes starting at 082f7170 (+/- 0)...
082f7170: 00000029 00000001 00000000 00000000
082f7180: 00000000 ffffffff 00000000 00000002
082f7190: 00000000 00000000 00000000 00000000
082f71a0: 00000000 00000000 00000000 00000000
082f71b0: ffffffff ffffffff 00000000 00000002
082f71c0: 00000001 00000000 00000001 00000003
082f71d0: 00000000 00000000 00000001 00000003
082f71e0: 00000000 00000000 00000000 00000001
082f71f0: 00000000 00000000 00000000 00000000
082f7200: 00000000 0000004b 00000000 0000000e
082f7210: 00000000 00000011 082f7170 000000a0
082f7220: 082f7160 000000a9 0000001b 00000001
082f7230: 00000000 00000000 00000000 ffffffff
082f7240: 00000000 00000001 00000000 00000000
082f7250: 00000000 00000000 00000000 00000000
082f7260: 00000000 00000000 ffffffff ffffffff

```

**0x082f7170:** Jackpot. This pattern looks very promising indeed. The reason for this is that if one notes the pattern, one will see that the next potential item being listed in this linked list has an index of **0x011b**, or 283. By using the technique described earlier to determine what the name of this item is, one finds that it's name is "torch". A quick check of Reveng's inventory reveals, indeed, he does have his trusty torch with him, four torches in fact! The first thing to do now is to identify the base address and size of the elements. Since these elements are in the bss, memgrep's heap enumeration technique can be used to make this process easier.

```

$ memgrep -p 'pidof adom' -e -a 0x082f7170
Found allocation unit for 082f7170: base is 082f7170, length is 160.
Total matched heap entries: 1

```

Well fancy that, the address that was found is actually the base address for that allocation. The structure for Reveng's current item list is within his grasp, but first one needs to validate the theory. To do this, one could do a number of things, but nothing is simpler than simply trying to change an item from one type to another. As has been established to this point, items appear to be referred to by their index into the "item array" that was found earlier. If one were to want to modify an item, all one would need do is change an item from one identifier to another, in theory. To validate this theory, one can try changing Reveng's "gnomish boots" into a torch, given that the identifiers for both items is known.

```

$ memgrep -p 'pidof adom' -r -a 0x082f7170 -t i,283

```

Replacing memory 082f7170 with 4 bytes of data...

A few Sim Sim Sala-bim's later and Reveng's gnomish boots have magically transformed into... a torch!

```
leather cap [+0, +0] [5a]
chain mail <-1, -3> [-3, +5] [400a]
-
short sword <+0, 1d6> [30a]
small shield [+2, +0] [50a]
-
-
torch [10a]
```

Victory is very close, but one last thing remains. In order to reliably find the list of the items in memory, considering it's dynamically allocated at load time, one must first try to find a place in memory, perhaps the data section, that refers to the first item in the list. This address would be useful as it would not change from allocation to allocation. There are two approaches for this method. First of all, one could attempt to manually walk the previous pointers in the linked list element, assuming there is one, until the items begin to loop. This method is tedious and error prone. The second method involves making use of memgrep's heap enumeration feature to look at memory allocations of the same size. This method is application dependant, so it's possible that it will not be as useful as the first method. For the purpose of discussion, the second method has been chosen. It is known that the size of the elements in the array is 160 bytes, as determined by the previous heap check. With this knowledge in hand, one can check the heap for 160 byte allocations. The address that is the lowest in memory should be the first item<sup>6</sup>.

```
$ memgrep -p 'pidof adom' -e -m 160 | grep 160
Heap entry: 082f7738 (size 160)
Heap entry: 082f7660 (size 160)
Heap entry: 082f7588 (size 160)
Heap entry: 082f74b0 (size 160)
Heap entry: 082f73d8 (size 160)
Heap entry: 082f7300 (size 160)
Heap entry: 082f7228 (size 160)
Heap entry: 082f7170 (size 160)
Heap entry: 082f70b8 (size 160)
Heap entry: 082f7000 (size 160)
Heap entry: 082f6f48 (size 160)
Heap entry: 082f6e90 (size 160)
```

Given that the lowest address is 0x082f6e90, that seems like the best place to start searching for references in the data and bss sections. An encouraging sign

<sup>6</sup>This is dependant on the memory manager being used.

is that all of the above allocations are relatively close to one another, lending credence to the fact that they were all allocated at the same time<sup>7</sup>.

```
$ memgrep -p 'pidof adom' -s -a bss -f x,906e2f08
Searching 0x082626a0 => 0xffffffff...
    found at 0x082a7cc0
    found at 0x082f6f38
```

Hmm, two references, both of which are in bss. This is not necessarily bad news, however. The bss includes uninitialized data. As such, the global variable that stores the pointer to the item list may in actuality be in the bss. The first step is to take a look at the memory at 0x082a7cc0 and see what it looks like.

```
$ memgrep -p 'pidof adom' -d -a 0x082a7cc0 -l 128
128 bytes starting at 082a7cc0 (+/- 0)...
082a7cc0: 082f6e90 00000000 082f6f48 00000000
082a7cd0: 00000000 082f7000 082f70b8 00000000
082a7ce0: 00000000 00000000 00000000 082f7170
082a7cf0: 00000000 00000000 00000000 00000000
082a7d00: 082f72e0 00000000 00000000 00000000
082a7d10: 00000000 00000000 00000000 00000000
082a7d20: 00000000 00000000 00000000 00000000
082a7d30: 00000000 00000000 00000000 00000000
```

Reading rainbow challenge: What display pattern does the above output match? The answer after a few quick... err, rather, it matches the display output shown on the inventory screen. Reveng currently has an item equipped in his Head slot. He does not have an item equipped in his Neck slot. He does have an item equipped in his body slot, and so on. To verify that this isn't just a red herring, one can verify that the item referred to by the 0x082f6e90 pointer (from the Head slot), is indeed a "leather cap".

```
$ memgrep -p 'pidof adom' -d -a 082f6e90 -l 4
4 bytes starting at 082f6e90 (+/- 0)...
082f6e90: 0000001c
```

```
$ memgrep -p 'pidof adom' -d -a 827473c -l 112
112 bytes starting at 0827473c (+/- 0)...
0827473c: 00000000 00000000 00000000 ffffffff
0827474c: 081b64d3 081b64c6 081b6757 081b6757
0827475c: 00000000 00000000 00000000 00000000
0827476c: 00000001 00000001 00000000 00000000
```

<sup>7</sup>The same time as in the phase of loading items from the save game file.

```
0827477c: 00000001 00000001 00000005 00000002
0827478c: 00000000 00000001 00000306 00000000
0827479c: 00000000 000007d0 00000003 ffffffff
```

```
$ memgrep -p 'pidof adom' -d -a 081b64d3 -c -l 32
32 bytes starting at 081b64d3 (+/- 0)...
081b64d3: leather cap.pair
081b64e3: s of heavy boots
```

Presto. The current characters item equipment lists can be found starting at **0x082a7cc0**. With that, one now has all the information needed to press on with figuring out exactly how items can be modified.

**Sidebar:** There are some other patterns not discussed in detail here that may be helpful with updating items in memory. The technique involves making use of the pattern that all items in memory follow the format:

$$[ \text{4 byte index} ] [ 0x00000001 ] [ 3 \times 0x00000000 ]$$

#### 4.4.1 Type

Changing the type of an item is fairly simple and was briefly discussed in the previous section. The process involves two things: finding the index of the item that is desired and finding the memory location of the item to be changed. The second task has been discussed in the previous section, and the first is discussed in great detail in the “Enumerating Items” section (4.6). With these two tactics firmly established one can combine them to effectively change any item from one to another.

The script that implements this functionality can be found in the appendix (A.4.1).

#### 4.4.2 Count

The count field associated with an item is the number of said items that the character has allocated to them. In the case of Reveng, he currently has 4 torches in his possession. This little bit of information will be useful in the process of identifying where exactly the number of items are located in the item structure, if at all.

In the previous section it was determined that the “torch” item’s structure can be found at 0x082f7228. The contents of the item are as follows:

```

$ memgrep -p 'pidof adom' -d -a 0x082f7228 -l 160
160 bytes starting at 082f7228 (+/- 0)...
082f7228: 000011b 00000001 00000000 00000000
082f7238: 00000000 ffffffff 00000000 00000001
082f7248: 00000000 00000000 00000000 00000000
082f7258: 00000000 00000000 00000000 00000000
082f7268: ffffffff ffffffff 00000000 00000000
082f7278: 00000000 00000000 00000001 00000004
082f7288: ffffffff c 00000000 00000001 00000003
082f7298: 00000000 00000000 00000000 00000004
082f72a8: 00000000 00000000 00000000 00000000
082f72b8: 00000000 0000004b 00000000 00000000

```

If one looks closely one will see that there are two integer values in the structure that are set to the number 4. In the interest of being a mad scientist running around with a ice pick, it seems only fitting that a trial and error method be used to try to determine which of the two, if either, affects the count of the item.

```

$ memgrep -p 'pidof adom' -r -a 0x082f7284 -t i,9
Replacing memory 082f7284 with 4 bytes of data...

```

And a quick check reveals that... no, this is not the address. As to what was set... who knows. Nothing obvious can be noted from the display output. Anyways, onto the second value.

```

$ memgrep -p 'pidof adom' -r -a 0x082f72a4 -t i,9
Replacing memory 082f72a4 with 4 bytes of data...

```

Bingo. Reveng now has 9 torches. Yippie! Now, it may not be too wise to save at this point since an unknown value was changed from 4 to 9, so a quick switch back will put things back in order. With that verification, one now knows that **0x7c** bytes into the character item structure holds the count of the given item.

The script that implements this functionality can be found in the appendix (A.4.2).

### 4.4.3 Weight

So, a little heavy set around the waste? Worry not! Weight reduction techniques are here to stay... or something. The easiest way to approach reducing the weight of an item is to start with an item that has a somewhat substantial weight. One such item is the iron ration weighing in at 100 stones. This is far too heavy for little Reveng, so assistance is thusly needed.



```
Food <'/>
F - iron ration [100s]
```

There are two potential places where the weight associated with an item could be stored, one being more likely than the other. The first and most likely place would be in the global item array found in the data section. This is most likely considering most, if not all, items will have a static weight. However, the weight could also be stored in the current character item list.

Following the path of likelihood, the first thing to do is identify where in the global item list the 'iron ration' item is located. This technique has been discussed previously and will thus be skipped in the interest of decreasing redundancy. In this case, the location of the iron ration item structure is 0x0827442c (index 21). Given this, a dump of the memory is order.

```
$ memgrep -p 'pidof adom' -d -a 0x0827442c -l 112
112 bytes starting at 0827442c (+/- 0)...
0827442c: 00000000 00000014 00000000 ffffffff
0827443c: 081b657f 081b6572 081b6757 081b6757
0827444c: 00000000 00000000 00000000 00000000
0827445c: 00000001 00000002 00000000 00000000
0827446c: 00000001 00000003 00000064 0000000a
0827447c: 00000000 00000001 00000000 00000000
0827448c: 00000000 00002710 00000007 ffffffff
```

Looks like this will be a piece of cake. **0x48** bytes into the structure one can note the occurrence of a 4-byte integer that is set to 100, just the number being searched for. Once again, assuming the role of the wild and crazy doctor with an oversized ice pick, a quick test to set the value to 47 reveals that, indeed, this is the weight field used by items. It's really as simple as that.

```
Food <'/>
F - iron ration [47s]
```

The script that implements this functionality can be found in the appendix (A.4.3).

#### 4.4.4 Dice

In order to separate one's self from from a crowd of mythical foe who wield the same weapon is to have that weapon be blessed with ungodly bonuses. One should know by now that Reveng is not the kind of gnome to settle for second best, and as such his Trident of the Red Rooster must be transformed into an awe inspiring site. The path taken to do this will involve the same logic as did the weight modification path. Items will either have their dice information

stored in the global item array in the data section or in the character's current item list. The thing to note about this, though, is that items with the same name can have different stats, such as different dice and attribute modifiers. This difference leads one to suspect that these fields are found not in the data section, but rather in the current item list. It is there where this leg of the quest will start.

```
Two-handed weapons <'>
n - Trident of the Red Rooster <*0, 1d2> <St+0>
```

From the start one knows that the Trident of the Red Rooster, at least the one Reveng has, defaults to 1d2 (one two-sided die) with no bonus. The 1d2 is the first thing to key off of as a possible form of identification. The most logical thing to do is to check the entry in the current item list and see if there are any 4-byte integer fields set to 1 and 2, respectively. It could also be the case that the dice are not stored in 4-byte integers, but rather single-bytes. However, that remains to be seen... for now.

```
$ memgrep -p 'pidof adom' -s -a bss \
-f x,9402000001000000000000000000000000000000000000000000000000000000
Searching 0x0829e5c0 => 0xffffffff...
found at 0x082f74b0
$ memgrep -p 'pidof adom' -d -a 0x082f74b0 -l 160
160 bytes starting at 082f74b0 (+/- 0)...
082f74b0: 00000294 00000001 00000000 00000000
082f74c0: 00000000 ffffffff 00000000 00000002
082f74d0: 00000000 00000000 00000000 00000000
082f74e0: 00000000 00000000 00000000 00000000
082f74f0: ffffffff ffffffff 00000000 00000000
082f7500: 00000000 00000000 00000001 00000002
082f7510: 00000000 00000000 00000001 00000002
082f7520: 00000000 00000000 00000000 00000001
082f7530: 00000000 00000000 00000000 00000000
082f7540: 00000000 0000004c 00000000 00000008
```

Hmm, there are a few fields set to 1 and 2. Making an educated guess, it's most likely that the numbers will be found in the order in which they are displayed, as that has been a common pattern throughout this analysis. However, that still leaves two possibilities. If in doubt, run around with scissors and see which one works. The first test will involve setting the memory at 0x082f7508 to the 4-byte integer 5 to see if the number of dice changes.

```
$ memgrep -p 'pidof adom' -r -a 0x082f7508 -t i,5
Replacing memory 082f7508 with 4 bytes of data...
```

```
Two-handed weapons <'>
â - Trident of the Red Rooster <+0, 5d0> <St+0>
```

Excellent, the number of dice is now five. However, for some reason the sides of the dice has now dropped to zero. Perhaps the number of sides of the dice is calculated based on the actual number of dice? If that's the case, one could try setting the memory 4 bytes after the number of dice to 15. If this theory is true, the display should show that the number of sides is 3, as derived from 15 divided by 5. Does this theory hold water?

```
$ memgrep -p 'pidof adom' -r -a 0x082f750c -t i,15
Replacing memory 0x082f750c with 4 bytes of data...
```

```
Two-handed weapons <'>
â - Trident of the Red Rooster <+0, 5d3> <St+0>
```

It holds **gallons** of water! The mystery has been unlocked, and as such one can now set the number of dice and their sides associated with an item at will. Some of the other fields in this structure will not be covered but can be used to apply bonus modifiers and other interesting things. Poke around a bit.

The script that implements this functionality can be found in the appendix (A.4.4).

## 4.5 Enumerating MOBs

**Techniques Used:** Known Value, Contiguous Patterns

Enumerating the list of MOBs, or Mobile Objects, in the game realm can be handy for a number of purposes. If one does extended research one will likely find a way to modify character attributes, such as their relative strength and hitpoints. This could turn Reveng's encounter, not that he isn't already the baddest gnome around, into a smooth waltz in the park. The actual method used to locate the list of MOBs in the game is quite similar to that used in item enumeration (4.6). Specifically, start with a known MOB name, such as a "large bat", and search rodata for places that his name occurs, preferably by itself.

```
$ memgrep -p 'pidof adom' -s -a rodata -f x,'echo -n "large bat" \
| xxd -ps -c 30'00
Searching 0x081507a0 => 0xffffffff...
found at 0x081a1496
```

Now, in theory the list of MOBs will likely be found in the data section considering the actual pool of MOBs in the game is static, not dynamic.

```
$ memgrep -p 'pidof adom' -s -a data -f x,96141a08
Searching 0x082626a0 => 0xffffffff...
    found at 0x082650d0
```

Indeed, one reference found in the data section. So far so good. Now it's time to analyze the memory and attempt to find a pattern. Due to the fact that a match was found in the data section, it's very likely that one will be dealing with an array of structures.

```
$ memgrep -p 'pidof adom' -d -a 0x082650d0 -l 196 -b 64
196 bytes starting at 082650d0 (+/- 64)...
08265090: 00000084 00000064 00000004 00000000
082650a0: 00000000 00800000 00000000 00800000
082650b0: 0000000a 0000016f 00000000 00000010
082650c0: 00000028 00000014 081a14a0 081a1712
082650d0: 081a1496 081a184d 00000042 00000001
082650e0: 0000000e 00000001 00000002 00000000
082650f0: 00000001 00000000 081a1490 00000001
08265100: 00000004 00000002 00000001 00000096
08265110: 00000006 00000020 00000080 00000064
08265120: 00000002 00000000 00000000 00000000
08265130: 00000000 00a00010 00000000 00000000
08265140: 00000000 0000000b 0000003c 0000000f
08265150: 081a1220 081a1712 081a11fd 081a184d
08265160: 00000072 00000001 0000000d 00000001
08265170: 00000001 ffffffff 00000001 00000000
08265180: 081a1490 00000001 00000005 00000006
08265190: 00000001 00000064 00000006 00000020
082651a0: 00000082 00000064 00000002 00000000
082651b0: 00000000 00400000 00000000 00000010
082651c0: 0000001e 00000000 00000000 0000000c
082651d0: 0000003c
```

That pattern looks pretty straight forward. In order to continue, however, one will need to find out where the pattern stops. If one moves down in memory in increments, the pattern should eventually stop, at which one point should be able to make a determination as to where the array starts.

```
$ memgrep -p 'pidof adom' -d -a 0x08264fb0 -l 384
384 bytes starting at 08264fb0 (+/- 0)...
08264fb0: b40bbe37 c30c8ea1 5a05df1b 2d02ef8d
08264fc0: 081a184f 081a184d 00000067 00000000
08264fd0: 0000000a 00000000 00000001 ffffffff
08264fe0: 00000001 ffffffff 081a1847 00000001
```

```

08264ff0: 00000008 00000008 00000001 00000096
08265000: 00000009 00000008 00000101 00000064
08265010: 00000004 00000000 00000000 00000000
08265020: 00000000 00000000 00000000 00000000
08265030: 00000000 00000006 00000032 0000000f
08265040: 081a1720 081a1712 081a170e 081a184d
08265050: 0000006f 00000000 0000000a 00000000
08265060: 00000001 00000001 00000001 00000000
08265070: 081a1708 00000002 00000008 00000008
08265080: 00000001 00000096 0000000a 00040008
08265090: 00000084 00000064 00000004 00000000
082650a0: 00000000 00800000 00000000 00800000
082650b0: 0000000a 0000016f 00000000 00000010
082650c0: 00000028 00000014 081a14a0 081a1712
082650d0: 081a1496 081a184d 00000042 00000001
082650e0: 0000000e 00000001 00000002 00000000
082650f0: 00000001 00000000 081a1490 00000001
08265100: 00000004 00000002 00000001 00000096
08265110: 00000006 00000020 00000080 00000064
08265120: 00000002 00000000 00000000 00000000

```

The misleading part about identifying the pattern in this instance is that the pointers to the name information for a given MOB are actually at the end of the structure, not the beginning. The two pointers at 0x08264fc0 are red herrings and aren't actually part of the MOB array. The actual start point for the array of MOBs is **0x08264fc8**. Doing a quick calculation, one determines that the size of the MOB structure is **0x88**, or 136 bytes. Finally, following memory up until the pattern once again changes, one can note that the last structure ends at **0x08272dd8**. The calculation to find the total number of elements in an array returns **418** total MOBs in the game, or at least in this array.

Enumerating the array is as simple as starting at the start address (0x08264fc8) and continually increment in blocks of 136 until the address end address (0x08272dd8) has been reached. The pointer to the name of a given MOB can be found 0x80 bytes into the structure.

The script that implements this design can be found in the appendix (A.5).

## 4.6 Enumerating Items

**Techniques Used:** Known Value, Contiguous Patterns

The process of enumerating items is something that can be inferred from the previously discussed sections, but it's always best to be thorough. In order to enumerate the list of available items in the game, one must know three things.

The first thing to know is where the item array starts at, which in this case is 0x08273afc. The second thing to know is how big each element in the array is, which in this case is 112 bytes. Lastly, one must know how many elements there are in the array, which in this case there are 697. With all this knowledge in hand, one need only try to decode some of the more important parts of the item structure. As was determined from research in previous sections, 16 bytes into the item structure one can find the pointers to 4 strings. The first string points to the name of the item, the second points to the plural name of the item, and the third and fourth strings are commonly just empty strings. This information can be rewritten in the form of a C structure like so:

```
struct _item
{
    char unknown[16];
    char *singularName;
    char *pluralName;
    char *unknown2;
    char *unknown3;
    char unknown[80];
};
```

For the purpose of enumeration, the rest of the fields in the structure are not important. Now that the structure is more clear, one can simply start at the base address for the array and work up element by element, inspecting the `singularName` attribute of the structure to see if it is the item desired. If not, continue on. Once the item desired is found, one can then calculate the index by subtracting the address of the item from the base address and dividing by the size of the element (112 bytes).

The script that implements this design can be found in the appendix (A.6).

## Chapter 5

# Conclusion

It is the author's hope that the reader now has a more refreshed or better understanding of how, and when, memory analysis can be used when dealing with black box applications. The concepts discussed in this document, while not ground-breaking, do indeed lend themselves well for general approaches that can be applied to reverse engineering outside of the normally pursued mediums.

# Appendix A

## Case Study Source Code

### A.1 Setting Character Attribute Levels

```
#!/usr/bin/perl

use strict;

my $base = 0x082b1728;

print "Maxing attributes...\n";

$cmd = sprintf("memgrep -p 'pidof adom' -r -a %x -t " .
               "x,6300000063000000630000006300000063000000" .
               "63000000630000006300000063000000" .
               "63000000630000006300000063000000" .
               "63000000630000006300000063000000",
               $base);

system("$cmd");

print "Finished.\n";
```



## A.2 Setting Character Experience

```
#!/usr/bin/perl

use strict;

my $base = 0x082b1700;
my $exp = $ARGV[0];

if (not defined($exp)) {
    $exp = 18651974;
}

print "Setting experience counter to $exp...\n";

my $cmd = sprintf("memgrep -p 'pidof adom' -r -a %x -t i,$exp",
    $base);

system("$cmd");

print "Finished.\n";
```

## A.3 Setting Skill Levels

```
#!/usr/bin/perl

use strict;

my $base = 0x082ac100;

print "Increasing and adding skills...\n";

my %skillTypeHash;

$skillTypeHash{0x00} = 0x05;    # Alchemy
$skillTypeHash{0x01} = 0x06;    # Alertness
$skillTypeHash{0x02} = 0x03;    # Appraising
$skillTypeHash{0x03} = 0x05;    # Archery
$skillTypeHash{0x04} = 0x08;    # Athletics
$skillTypeHash{0x05} = 0x09;    # Backstabbing
$skillTypeHash{0x06} = 0x06;    # Bridge building
$skillTypeHash{0x07} = 0x04;    # Climbing
$skillTypeHash{0x08} = 0x09;    # Concentration
$skillTypeHash{0x09} = 0x07;    # Cooking
$skillTypeHash{0x0a} = 0x04;    # Courage
$skillTypeHash{0x0b} = 0x09;    # Detect item status
$skillTypeHash{0x0c} = 0x08;    # Detect traps
$skillTypeHash{0x0d} = 0x09;    # Disarm traps
$skillTypeHash{0x0e} = 0x06;    # Dodge
$skillTypeHash{0x0f} = 0x05;    # Find weakness
$skillTypeHash{0x10} = 0x09;    # First aid
$skillTypeHash{0x11} = 0x03;    # Fletchery
$skillTypeHash{0x12} = 0x08;    # Food preservation
$skillTypeHash{0x13} = 0x06;    # Gardening
$skillTypeHash{0x14} = 0x07;    # Gemology
$skillTypeHash{0x15} = 0x08;    # Hagglng
$skillTypeHash{0x16} = 0x09;    # Healing
$skillTypeHash{0x17} = 0x08;    # Herbalism
$skillTypeHash{0x18} = 0x06;    # Law
$skillTypeHash{0x19} = 0x04;    # Listening
$skillTypeHash{0x1a} = 0x09;    # Literacy
$skillTypeHash{0x1b} = 0x04;    # Metallurgy
$skillTypeHash{0x1c} = 0x05;    # Mining
$skillTypeHash{0x1d} = 0x06;    # Music
$skillTypeHash{0x1e} = 0x09;    # Necromancy
$skillTypeHash{0x1f} = 0x09;    # Pick locks
$skillTypeHash{0x20} = 0x08;    # Pick pockets
```

```

$skillTypeHash{0x21} = 0x09;    # Smithing
$skillTypeHash{0x22} = 0x03;    # Sealth
$skillTypeHash{0x23} = 0x08;    # Survival
$skillTypeHash{0x24} = 0x04;    # Swimming
$skillTypeHash{0x25} = 0x04;    # Tactics
$skillTypeHash{0x26} = 0x07;    # Two weapon combat
$skillTypeHash{0x27} = 0x08;    # Ventriloquism
$skillTypeHash{0x28} = 0x06;    # Woodcraft

foreach my $x (0..39)
{
    my $curr = $base + 20 * $x;
    my $skill = $x + 1;
    my $cmd;

    $cmd = sprintf("mimgrep -p 'pidof adom' -r -a %x -t x," . "%.2x" .
                    "00000001000000000000000000000000000064" . "%.2x" . "ff00",
                    $curr, $skill, $skillTypeHash{$skill});

    system("$cmd");
}

print "Finished.\n";

```

## A.4 Changing Item Attributes

### A.4.1 Type

```
#!/usr/bin/perl

use strict;

my $itemArrayBase = 0x08273afc;
my $numItems      = 697;
my $elementSize   = 112;
my $fromItemName  = $ARGV[0] or die "No from item name supplied.";
my $toItemName    = $ARGV[1] or die "No to item name supplied.";
my $pid           = 'pidof adom';

chomp($pid);

my $fromItemIndex = findItemIndex(itemName => $fromItemName);
my $toItemIndex   = findItemIndex(itemName => $toItemName);

# Is the index valid?
if (($fromItemIndex == $numItems) or
    ($toItemIndex == $numItems)) {
    print "One of the items supplied was not found.\n";
    exit;
}

print "Changing item '$fromItemName' ($fromItemIndex)" .
      " to '$toItemName' ($toItemIndex)...\n";

my $cmd = sprintf("memgrep -p $pid -sr -a bss" .
                  " -f x,%.2x%.2x00000100000000000000000000000000" .
                  " -t x,%.2x%.2x00000100000000000000000000000000",
                  $fromItemIndex & 0xff, ($fromItemIndex >> 8) & 0xff,
                  $toItemIndex & 0xff, ($toItemIndex >> 8) & 0xff);

system($cmd);

print "Item change completed.\n";

sub findItemIndex
{
    my ($itemName) = @{{@_}}{qw/itemName/};
    my $current    = $itemArrayBase;
    my $x          = 0;

```

```

# Gold does not have an actual name
if (lc($itemName) eq 'gold') {
    return 37;
}

while ($x < $numItems)
{
    my ($addr, $name, $cmd, $buf);
    my @names;

    # Get the address of the string from the item.
    $cmd = sprintf("memgrep -p $pid -d -a 0x%.8x -l 4 | \
        tail -1 | cut -f 2 -d :", $current + 16);
    $addr = '$cmd';
    $addr =~ s/\W//g;

    # Now get the string
    $cmd = sprintf("memgrep -p $pid -d -a 0x%.8x -l 256 -c | \
        grep -v starting", hex($addr));
    $buf = '$cmd';
    $buf =~ s/^(.+: )//gm;
    $buf =~ s/\n//gm;
    @names = split /\./, $buf;
    $name = ((scalar(@names)) ? $names[0] : "unknown");

    if ($name eq $itemName) {
        last;
    }

    $current += $elementSize;
    $x++;
}

return $x;
}

```

## A.4.2 Count

```
#!/usr/bin/perl

use strict;

my $itemArrayBase = 0x08273afc;
my $numItems      = 697;
my $elementSize   = 112;
my $itemName      = $ARGV[0] or die "No item name supplied.";
my $itemCount     = $ARGV[1] or die "No new item count supplied.";
my $pid           = 'pidof adom';
my @addrs;
my $buf;

chomp($pid);

print "Changing count of item '$' . $itemName . "'...\n";

my $itemIndex = findItemIndex(itemName => $itemName);

# Is the index valid?
if ($itemIndex == $numItems) {
    print "The item supplied was not found.\n";
    exit;
}

my $cmd = sprintf("memgrep -p $pid -s -a bss -f" .
                  "x,%.2x%.2x00000100000000000000000000000000 |" .
                  "grep found",
                  $itemIndex & 0xff, ($itemIndex >> 8) & 0xff);

$buf = '$cmd';
$buf =~ s/found at//gm;
$buf =~ s/(\s+)(0x.+)(\s+)/$2 /gm;
$buf =~ s/\n/ /gm;
@addrs = split / /, $buf;
$buf = "";

foreach my $addr (@addrs) {
    my $a = sprintf("%.8x", hex($addr) + 0x7c);

    if (length($buf) == 0) {
        $buf = $a;
    } else {
        $buf .= ",$a";
    }
}
```

```

    }
}

system("memgrep -p $pid -r -a $buf -t i,$itemCount");

print "Item count successfully changed at " . scalar(@addrs) .
      " addresses.\n";

sub findItemIndex
{
    my ($itemName) = @{{@_}}{qw/itemName/};
    my $current = $itemArrayBase;
    my $x       = 0;

    # Gold does not have an actual name
    if (lc($itemName) eq 'gold') {
        return 37;
    }

    while ($x < $numItems)
    {
        my ($addr, $name, $cmd, $buf);
        my @names;

        # Get the address of the string from the item.
        $cmd = sprintf("memgrep -p $pid -d -a %.8x -l 4 | \
                        tail -1 | cut -f 2 -d :", $current + 16);
        $addr = '$cmd';
        $addr =~ s/\W//g;

        # Now get the string
        $cmd = sprintf("memgrep -p $pid -d -a 0x%.8x -l 256 -c | \
                        grep -v starting", hex($addr));
        $buf = '$cmd';
        $buf =~ s/^(.+: )//gm;
        $buf =~ s/\n//gm;
        @names = split /\./, $buf;
        $name = ((scalar(@names)) ? $names[0] : "unknown");

        if ($name eq $itemName) {
            last;
        }

        $current += $elementSize;
        $x++;
    }
}

```

```
    return $x;  
}
```



### A.4.3 Weight

```
#!/usr/bin/perl

use strict;

my $itemArrayBase = 0x08273afc;
my $numItems      = 697;
my $elementSize   = 112;
my $itemName      = $ARGV[0] or die "No item name supplied.";
my $itemWeight    = $ARGV[1] or die "No new item weight supplied.";
my $pid           = 'pidof adom';

chomp($pid);

print "Changing weight of item '" . $itemName . "'...\n";

my $itemAddress = findItemAddress(itemName => $itemName);

# Is the item valid?
if (((($itemAddress - $itemArrayBase) / $elementSize) == $numItems) {
    print "The item supplied was not found.\n";
    exit;
}

$itemAddress += 0x48;

my $cmd = sprintf("memgrep -p $pid -r -a %.8x -t i,$itemWeight",
    $itemAddress);

system($cmd);

print "Item weight successfully changed to $itemWeight.\n";

sub findItemAddress
{
    my ($itemName) = @{{@_}}{qw/itemName/};
    my $current = $itemArrayBase;
    my $x       = 0;

    # Gold does not have an actual name
    if (lc($itemName) eq 'gold') {
        return 37;
    }
}
```

```

while ($x < $numItems)
{
    my ($addr, $name, $cmd, $buf);
    my @names;

    # Get the address of the string from the item.
    $cmd = sprintf("memgrep -p $pid -d -a %.8x -l 4 | \
                    tail -1 | cut -f 2 -d :", $current + 16);
    $addr = '$cmd';
    $addr =~ s/\W//g;

    # Now get the string
    $cmd = sprintf("memgrep -p $pid -d -a 0x%.8x -l 256 -c | \
                    grep -v starting", hex($addr));
    $buf = '$cmd';
    $buf =~ s/^(.+: )//gm;
    $buf =~ s/\n//gm;
    @names = split /\./, $buf;
    $name = ((scalar(@names)) ? $names[0] : "unknown");

    if ($name eq $itemName) {
        last;
    }

    $current += $elementSize;
    $x++;
}

return $current;
}

```

## A.4.4 Dice

```
#!/usr/bin/perl

use strict;

my $itemArrayBase = 0x08273afc;
my $numItems      = 697;
my $elementSize   = 112;
my $itemName      = $ARGV[0] or die "No item name supplied.";
my $numDice       = $ARGV[1] or die "No dice count supplied.";
my $numSides      = $ARGV[2] or die "No dice sides supplied.";
my $pid           = 'pidof adom';
my $cmd;

chomp($pid);

my $itemIndex = findItemIndex(itemName => $itemName);

# Is the index valid?
if ($itemIndex == $numItems) {
    print "The item supplied was not found.\n";
    exit;
}

print "Changing item dice to " . $numDice . "d" . $numSides . "... \n";

my $itemAddress = findItemListAddress(itemIndex => $itemIndex);

if (length($itemAddress) == 0) {
    print "The item could not be found in the characters item list.\n";
    exit;
}

my $diceOffset = $itemAddress + 0x58;
my $sidesOffset = $itemAddress + 0x5c;

$numSides *= $numDice;

$cmd = sprintf("memgrep -p $pid -r -a %.8x -t i,$numDice",
               $diceOffset);
system($cmd);

$cmd = sprintf("memgrep -p $pid -r -a %.8x -t i,$numSides",
               $sidesOffset);
```

```

system($cmd);

print "Item dice change completed.\n";

sub findItemIndex
{
    my ($itemName) = @{{@_}}{qw/itemName/};
    my $current = $itemArrayBase;
    my $x      = 0;

    # Gold does not have an actual name
    if (lc($itemName) eq 'gold') {
        return 37;
    }

    while ($x < $numItems)
    {
        my ($addr, $name, $cmd, $buf);
        my @names;

        # Get the address of the string from the item.
        $cmd = sprintf("memgrep -p $pid -d -a %.8x -l 4 | \
            tail -1 | cut -f 2 -d :", $current + 16);
        $addr = '$cmd';
        $addr =~ s/\W//g;

        # Now get the string
        $cmd = sprintf("memgrep -p $pid -d -a 0x%.8x -l 256 -c | \
            grep -v starting", hex($addr));
        $buf = '$cmd';
        $buf =~ s/^(.+: )//gm;
        $buf =~ s/\n//gm;
        @names = split /\./, $buf;
        $name = ((scalar(@names)) ? $names[0] : "unknown");

        if ($name eq $itemName) {
            last;
        }

        $current += $elementSize;
        $x++;
    }

    return $x;
}

```



## A.5 Enumerating MOBs

```
#!/usr/bin/perl

use strict;

my $itemBase    = 0x08264fc8;
my $current     = $itemBase;
my $elementSize = 136;
my $numItems    = 418;
my $pid         = 'pidof adom';
my $x           = 0;

chomp($pid);

print "Enumerating MOBs...\n";

while ($x < $numItems)
{
    my $nameOffset = $current + 128;
    my $nameAddr;
    my $name;
    my $cmd;

    # Find the address of the name
    $cmd = sprintf("memgrep -p $pid -d -a %.8x -l 4 | \
                    tail -1 | cut -f 2 -d :", $nameOffset);
    $nameAddr = '$cmd';
    $nameAddr =~ s/\W//g;

    # Extract the name from the address
    $name = extractMobName(addr => hex($nameAddr));

    print "index $x: $name\n";

    $current += $elementSize;
    $x++;
}

print "Finished.\n";

sub extractMobName
{
    my ($addr) = @_{@_}{qw/addr/};
    my @names;
```

```
my $name;
my $cmd;
my $buf;

$cmd = sprintf("memgrep -p $pid -d -a 0x%.8x -l 256 -c | \
               grep -v starting", $addr);
$buf = '$cmd';
$buf =~ s/^(.+: )//gm;
$buf =~ s/\n//gm;
@names = split /\./, $buf;
$name = ((scalar(@names)) ? $names[0] : "unknown");

return $name;
}
```

## A.6 Enumerating Items

```
#!/usr/bin/perl

use strict;

my $itemBase    = 0x08273afc;
my $current     = $itemBase;
my $elementSize = 112;
my $numItems    = 697;
my $pid        = 'pidof adom';
my $x          = 0;

chomp($pid);

print "Enumerating items...\n";

while ($x < $numItems)
{
    my $nameOffset = $current + 16;
    my $nameAddr;
    my $name;
    my $cmd;

    # Find the address of the name
    $cmd = sprintf("memgrep -p $pid -d -a %.8x -l 4 | \
                    tail -1 | cut -f 2 -d :", $nameOffset);
    $nameAddr = '$cmd';
    $nameAddr =~ s/\W//g;

    # Extract the name from the address
    $name = extractItemName(addr => hex($nameAddr));

    print "index $x: $name\n";

    $current += $elementSize;
    $x++;
}

print "Finished.\n";

sub extractItemName
{
    my ($addr) = @_{@_}{qw/addr/};
    my @names;
```



```
my $name;
my $cmd;
my $buf;

$cmd = sprintf("memgrep -p $pid -d -a 0x%.8x -l 256 -c | \
               grep -v starting", $addr);
$buf = '$cmd';
$buf =~ s/^(.+: )//gm;
$buf =~ s/\n//gm;
@names = split /\./, $buf;
$name = ((scalar(@names)) ? $names[0] : "unknown");

return $name;
}
```