Context-keyed Payload Encoding

Preventing Payload Disclosure via Context

October, 2007

I)ruid, C^{2}ISSP
<druid@caughq.org>
http://druid.caughq.org

Abstract

A common goal of payload encoders is to evade a third-party detection mechanism which is actively observing attack traffic somewhere along the route from an attacker to their target, filtering on commonly used payload instructions. The use of a payload encoder may be easily detected and blocked as well as opening up the opportunity for the payload to be decoded for further analysis. Even so-called keyed encoders utilize easily observable, recoverable, or guessable key values in their encoding algorithm, thus making decoding on-the-fly trivial once the encoding algorithm is identified. It is feasible that an active observer may make use of the inherent functionality of the decoder stub to decode the payload of a suspected exploit in order to inspect the contents of that payload and make a control decision about the network traffic. This paper presents a new method of keying an encoder which is based entirely on contextual information that is predictable or known about the target by the attacker and constructible or recoverable by the decoder stub when executed at the target. An active observer of the attack traffic however should be unable to decode the payload due to lack of the contextual keying information.

1 Introduction

In the art of vulnerability exploitation there are often numerous hurdles that one must overcome. Examples of hurdles can be seen as barriers to traversing the attack vector and challenges with developing an effective vulnerability exploitation technique. A critical step in the later inevitabley requires the use of an exploit *payload*, traditionally referred to as *shellcode*. A payload is the functional exploit component that implements the exploit's purpose[1].

One barrier to successful exploitation may be that including certain byte values in the payload will not allow the payload to reach its destination in an executable form[2], or even at all. Another hurdle to overcome may be that an in-line network security monitoring device such as an Intrusion Prevention System (IPS) could be filtering network traffic for the particular payload that the exploit intends to deliver[3, 288–289], or otherwise extracting the payload for further automated analysis[4][5, 2]. Whatever the hurdle may be, many challenges relating to the payload portion of the exploit can be overcome by employing what is known as a *payload encoder*.

1.1 Payload Encoders

Payload encoders provide the utility of obfuscating the exploit's payload while it is in transit. Once the payload has reached its target, the payload is decoded prior to execution on the target system. This allows the payload to bypass various controls and restrictions of the type mentioned previously while still remaining in an executable form. In general, an exploit's payload will be encoded prior to packaging in the exploit itself and what is known as a *decoder stub* will be prepended to the encoded payload which produces a new, slightly larger payload. This new payload is then packaged within the exploit in favor of the original.

1.1.1 Encoder

The *encoder* can take many forms and provide its function in a number of different ways. At its most basic definition, an encoder is simply a function used when packaging a payload for use by an exploit which encodes the payload into a different form than the original. There are many different encoders available today, some of which provide encoding such as alphanumeric mixed-case text[6], Unicode safe mix-cased text[7], UTF-8 and tolower() safe[2], and XOR against a 4-byte key[8]. There is also an extremely impressive polymorphic XOR additive feedback encoder available called Shikata Ga Nai[9].

1.1.2 Decoder Stub

The *decoder stub* is a small chunk of instructions that is prepended to the encoded payload. When this new payload is executed on the target system, the decoder stub executes first and is responsible for decoding the original payload data. Once the original payload data is decoded, the decoder stub passes execution to the original payload. Decoder stubs generally perform a reversal of the encoding function, or in the case of an XOR obfuscation encoding, simply perform the XOR again against the same key value.

1.1.3 Example: Metasploit Alpha2 Alphanumeric Mixedcase Encoder (x86)

The Metasploit[?] Alpha2 Alphanumeric Mixedcase Encoder[6] encodes payloads as alphanumeric mixedcase text using SkyLined's Alpha2 encoding suite. This allows a payload encoded with this encoder to traverse such attack vectors as may require input to pass text validation functions such as the C89 standard functions `isalnum()` and `isprint()`, as well as the C99 standard function `isascii()`.

1.1.4 Keyed Encoders

Many encoders utilize encoding techniques which require a key value. The Call+4 Dword XOR encoder[8] and the Shikata Ga Nai polymorphic XOR additive feedback encoder[9] are examples of keyed encoders.

Key Selection

Encoders which make use of key data during their encoding process have traditionally used either random or static data chosen at the time of encoding, or data that is tied to the encoding process itself[10], such as the index value of the current position in the buffer being operated on, or a value relative to that index.

Example: Metasploit Single-byte XOR Countdown Encoder (x86)

The Metasploit Single-byte XOR Countdown Encoder[10] uses the length of the remaining payload to be operated upon as a position-dependent encoder key. The benefit that this provides is a smaller decoder stub, as the decoder stub does not need to contain any static keying information. Instead, it tracks the length property of the payload as it decodes and uses that information as the key.

Weaknesses

The most significant weakness of most keyed encoders available today is that the keying information that is used is either observable directly or constructable from the observed decoder stub. Either the static key information is transmitted within the exploit as part of the decoder stub itself, or the key information is reproducible once the encoding algorithm is known. Knowledge of the encoding algorithm is usually obtainable by recognizing known decoder stubs or analyzing unknown decoder stubs instructions in detail.

The expected inherent functionality of the decoder stub also introduces a weakness. Modern payload encoders rely upon the decoder stub's ability to properly decode the payload at run-time. It is feasible that an active observer may exploit this inherent functionality to decode a suspected payload within a sandbox environment in real-time[5, 3] in order to inspect the contents of the payload and make a control decision about the network traffic it was found in. Because the decoder stub requires only that it is being executed by a processor that will understand its instruction-set, producing such a sandbox is trivial.

Unfortunately, all of the aforementioned keyed encoders include the static key value directly in their decoder stubs and are thus vulnerable to the weaknesses described here. This allows an observer of the encoded payload in transit to potentially decode the payload and inspect it's content. Fortunately, all of the keyed encoders previously mentioned could potentially be improved to use contextual keying as is described in the following chapter.

2 Contextual Keying

Contextual keying is defined as the process of selecting an encoding key from context information that is either known or predictable about the target. A *context-key* is defined as the result of that process. The context information available about the exploit's target may contain any number of various types of information, dependent upon the attacker's proximity to the target, knowledge of the target's operation or internals, or knowledge of the target's environment.

2.1 Encoder

When utilizing a context-key, the method of encoding is largely unchanged from current methods. The exploit crafter simply passes the encoding function the context-key as its static key value. The size of the context-key is dependent upon the requirements of the encoder being used; however, it is feasible that the key may be of any fixed length, or ideally the same size as the payload being encoded.

2.2 Decoder Stub

The decoder stub that requires a context-key is not only responsible for decoding the encoded payload but is also responsible for retrieving or otherwise generating its context-key from the information that is available to it at run-time. This may include retrieving a value from a known memory address, performing some calculation on other information available to it, or any number of other possible scenarios. The following section will explore some of the possibilities.

2.3 Application Specific Keys

2.3.1 Static Application Data

If the attacker has the convenience of reproducing the operating environment and execution of the target application, or even simply has access to the application's executable, a context-key may be chosen from information known about the address space of the running process. Known locations of static values such as environment variables, global variables and constants such as version strings, help text, or error messages, or even the application's instructions or linked library instructions themselves may be chosen from as contextual keying information.

Profiling the Application

To successfully select a context-key from a running application's memory, the application's memory must first be profiled. By polling the application's address space over a period of time, ranges of memory that change can be eliminated from the potential context-key data pool. The primary requirement of viable data in the process's memory space is that it *does not change over time* or between subsequent instantiations of the running application. After profiling is complete, the resultant list of memory addresses and static data will be referred to as the application's *memory map*.

Memory Map Creation

The basic steps to create a comprehensive memory map of a running process are:

1. Attach to the running process.
2. Initialize the memory map with a poll of non-null bytes in the running process's virtual memory.
3. Wait an arbitrary amount of time.
4. Poll the process's virtual memory again.
5. Find the differential between the contents of the memory map and the most recent memory poll.
6. Eliminate any data that has changed between the two from the memory map.
7. Optionally eliminate any memory ranges shorter than your desired key length.
8. Go to step 3.

Continue the above process until changing data is no longer being eliminated and store the resulting memory map as a map of that instance of the target process. Restart the application and repeat the above process, producing a second memory map for the second instance of the target process. Compare the two memory maps for differences and again eliminate any data that differs. Repeat this process until changing data is no longer being eliminated.

The resulting final memory map for the process must then be analyzed for static data that may be directly relative to the environment of the process and may not be consistent across processes running within different environments such as on different hosts or in different networks. This type of data includes network addresses and ports, host names, operating system "unames", and so forth. This type of data may also include installation paths, user names, and other user-configurable options during installation of the application. This type of data does *not* include application version strings or other pertinent information which may be directly relative to the properties of the application which contribute to the application being vulnerable and successfully exploited.

Identifying this type of information relative to the application's environment will produce two distinct types of memory map data; one type containing static application context data, and the other type containing environment context data. Both of these types of data can be useful as potential context-key values, however, the former will be more portable amongst targets whereas the latter will only be useful when selecting key values for the actual target process that was actively profiled. If it is undesirable, introducing instantiation of processes being profiled on different network hosts and with different installation configuration options to the memory map generation process outlined above will likely eliminate the latter from the memory map entirely.

Finally, the memory maps can be trimmed of any remaining NULL bytes to reduce their size. The final memory map should consist of records containing memory addresses and the string of static data which can be found in memory at those locations.

Memory Map Creation Methods

Metasploit Framework's msfpescan

One method to create a memory map of viable addresses and values is to use a tool provided by the Metasploit Framework called `msfpescan`. `msfpescan` is designed to scan PE formatted executable files and return the requested portion of the `.text` section of the executable. Data found in the `.text` section is useful as potential context-key data as the `.text` section is marked read-only when mapped into a process' address space and is therefore static and will not change. Furthermore, `msfpescan` predicts where in the executed process' address space these static values will be located, thus providing both the static data values as well as the addresses at which those values can be retrieved.

To illustrate, suppose a memory map for the Windows *System* service needs to be created for exploitation of the vulnerability described in Microsoft Security Bulletin MS06-040[11] by an exploit which will employ a context-keyed payload encoder. A common DLL that is linked into the service's executable when compiled can be selected as the target for msfpescan. In this case, `ws2help.dll` is chosen due to its lack of updates since August 23rd, 2001. Because this particular DLL has remained unchanged for over six years, its instructions provide a particularly consistent cache of potential context-keys for an exploit targeting an application linked against it anytime during the last six years. A scan of the first 1024 bytes of `ws2help.dll`'s executable instructions can be performed by executing the following command:

```
msfpescan -b 0x0 -A 1024 ws2help.dll
```

Furthermore, `msfpescan` has been improved via this research effort to render data directly as a memory map. This improved version is available in the Metasploit Framework as of version 3.1. A scan and dump to memory map of `ws2help.dll`'s executable instructions can be performed by executing the following command:

```
msfpescan --context-map context ws2help.dll
```

It is important to note that this method of memory map generation is much less comprehensive than the method previously outlined; however, when targeting a process whose executable is relatively large and links in a large number of libraries, profiling only the instruction portions of the executable and library files involved may provide an adequately-sized memory map for context-key selection.

Metasploit Framework's memdump.exe

The Metasploit Framework also provides another useful tool for the profiling of a running process' memory called `memdump.exe`. `memdump.exe` is used to dump the entire memory space of a running process. This tool can be used to provide the polling step of the memory map creation process previously outlined. By producing multiple memory dumps over a period of time, the dumps can be compared to isolate static data.

smem-map

A tool for profiling a Linux process' address space and creating a memory map is provided by this research effort. The `smem-map` tool[12] was created as a reference implementation of the process outlined at the beginning of this section. `smem-map` is a Linux command-line application and relies on the proc filesystem as an interface to the target process' address space.

The first time `smem-map` is used against a target process, it will populate an initial memory map with all non-null bytes currently found in the process's virtual memory. Subsequent polls of the memory ranges that were initially identified will eliminate data that has changed between the memory map and the most recent poll of the process's memory. If the tool is stopped and restarted and the specified memory map file exists, the file will be reloaded as the memory map to be compared against instead of populating an entirely new memory map. Using this functionality, a memory map can be refined over multiple sessions of the tool as well as multiple instantiations of the target process. A scan of a running process' address space can be performed by executing the following command:

```
smem-map <PID> output.map
```

Context-Key Selection

Once a memory map has been created for the target application, the encoder may select any sequential data from any memory address within the memory map which is both large enough to fill the desired key length and also does not produce any disallowed byte values in the encoded payload as defined by restrictions to the attack vector for the vulnerability. The decoder stub should then retrieve the context-key from the same memory address when executed at the target. If the decoder stub is developed so that it may read individual bytes of data from different locations, the encoder may select individual bytes from multiple addresses in the memory map. The encoder must note the memory address or addresses at which the context-key is read from the memory map for inclusion in the decoder stub.

Proof of Concept: Improved Shikata ga Nai

The Shikata ga Nai encoder[9], included with the Metasploit Framework, implements polymorphic XOR additive feedback encoding against a four byte key. The decoder stub that is prepended to a payload which has been encoded by Shikata ga Nai is generated based on dynamic instruction substitution and dynamic block ordering. The registers used by the decoder stub instructions are also selected dynamically when the decoder stub is constructed.

Improving the original Metasploit implementation of Shikata ga Nai to use contextual keying was fairly trivial. Instead of randomly selecting a four byte key prior to encoding, a key is instead chosen from a supplied memory map. Furthermore, when generating the decoder stub, the original implementation used a "`mov reg, val`" instruction (`0xb8`) to move the key value directly from its location in the decoder stub into the register it will use for the XOR operation. The context-key version instead uses a "`mov reg, [addr]`" instruction (`0xa1`) to retrieve the context-key from the memory location at *[addr]* and store it in the same register. The update to the Shikata ga Nai decoder stub was literally as simple as changing one instruction, and providing that instruction with the context-key's location address rather than a static key value directly.

The improved version of Shikata ga Nai described here is provided by this research effort and is available in the Metasploit Framework as of version 3.1. It can be utilized as follows from the Metasploit Framework Console command-line, after the usual exploit and payload commands:

```
set ENCODER x86/shikata_ga_nai
set EnableContextEncoding 1
set ContextInformationFile <application.map>
exploit
```

Case Study: MS04-007 vs. Windows XP SP0

The Metasploit framework currently provides an exploit for the vulnerability described in Microsoft Security Bulletin MS04-007[13]. The vulnerable application in this case is the Microsoft ASN.1 Library.

Before any exploitation using contextual keying can take place, the vulnerable application must be profiled. By opening the affected library from Windows XP Service Pack 0 in a debugger, a list of libraries that it itself includes can be gleaned. By collecting said library DLL files from the target vulnerable system, or an equivalent system in the lab, `msfpescan` can then be used to create a memory map:

```
msfpescan --context-map context \
   ms04-007-dlls/*
cat context/* >> ms04-007.map
```

After the memory map has been created, it can be provided to Metasploit and Shikata ga Nai to encode the payload that Metasploit will use to exploit the vulnerable system:

```
use exploit/windows/smb/ms04-007-killbill
set PAYLOAD windows/shell_bind_tcp
set ENCODER x86/shikata_ga_nai
set EnableContextEncoding 1
set ContextInformationFile ms04-007.map
exploit
```

2.3.2 Event Data

Similar to the static application data approach, transient data may also be used as a context-key so long as it persists long enough for the decoder stub to access it. Consider the scenario of a DNS server which is vulnerable to an overflow when parsing an incoming host name or address look-up request. If portions of the request are stored in memory prior to the vulnerability being triggered, the data provided by the request could potentially be used for contextual keying if it's location is predictable. Values such as IP addresses, port numbers, packet sequence numbers, and so forth are all potentially viable for use as a context-key.

2.3.3 Supplied Data

Similar to Event Data, an attacker may also be able to supply key data for later use to the memory space of the target application prior to exploitation. Consider the scenario of a caching HTTP proxy that exhibits the behavior of keeping recently requested resources in memory for a period of time prior to flushing them to disk for longer-term storage. If the attacker is aware of this behavior, the potential exists for the attacker to cause the proxy to retrieve a malicious web resource which contains a wealth of usable context-key data. Even if the attacker cannot predict where in memory the data may be stored, by having control of the data that is being stored other exploitation techniques such as *egg hunting*[14, 9][15] may be used by a decoder-stub to locate and retrieve context-key information when its exact location is unknown.

2.4 Temporal Keys

The concept of a *temporal address* was previously introduced by the paper entitled *Temporal Return Addresses: Exploitation Chronomancy*[16, 3]. In summary, a temporal address is a location in memory which holds timer data of some form. Potential types of timer data stored at a temporal address include such data as the system date and time, number of seconds since boot, or a counter of some other form.

The research presented in the aforementioned paper focused on leveraging the timer data found at such addresses as the return address used for vulnerability exploitation. As such, the viability of the data found at the temporal address was constrained by two properties of the data defined as *scale*, and *period*. These two properties dictate the window of time during which the data found at the temporal address will equate to the desired instructions. Another potential constraint for use of a temporal address as an exploit return address stems from the fact that the value contained at the temporal address is called directly for use as an executable instruction. If the memory range it is contained within is marked as non-executable such as with the more recent versions of Windows[16, 19], attempting use in this manner will cause an exception.

For the purpose that temporal addresses will be employed here, such strict constraints as those previously mentioned do not exist. Rather, the only desired property of the data stored at the temporal address which will be used as a context-key is that it does not change, or as in the case of temporal data, does not change during the time window in which we intend to use it. Due to this difference in requirements, the actual content of the temporal address is somewhat irrelevant and therefore is not constrained to a time-window in either the future or the past during which the data found at the temporal address will be fit for purpose. The viable time-window in the case of use for contextual keying is entirely constrained by duration rather than location along the time-line. Due to the values at different byte offsets within data found at a temporal address having differing update frequencies, selection of key data from these values produces varying duration time-windows during which the values will remain constant. By using single byte, dual byte, or otherwise relatively short context-keys, and carefully selecting from the available byte values stored within the timer found at the temporal address, the viable time-window chosen can be made to be quite lengthy.

2.4.1 Context-Key Selection

Provided by the previously mentioned temporal return address research effort is a very useful tool called `telescope`[16, 8]. The tool's function is to analyze a running process' memory for potential temporal addresses and report them to the user. By using this tool, potential context-key values and the addresses at which they reside can be respectively predicted and identified.

The temporal return addresses paper also revealed a section of memory that is mapped into all processes running on Windows NT, or any other more recent Windows system, called `SharedUserData`[16, 17]. The interesting properties of the `SharedUserData` region of a process' address space is that it is always mapped into memory at a predictable location and is required to be backwards compatible with previous versions. As such, the individual values contained within the region will always be at the same offset to it's predictable base address. One of the values contained within this region of memory is the system time, which will be used in the examples to follow.

Remotely Determining Time

Methods and techniques for profiling a target system's current time is outside of the scope of this paper, however the aforementioned paper on temporal return addresses[16, 13–15] offers some insight. Once a target system's current time has been identified, the values found at various temporal addresses in memory can be readily predicted to varying degrees of accuracy.

Time-Window Selection

It is important to note that when using data stored at a temporal address as a context-key, parts of that value are likely to be changing frequently. Fortunately, the key length being used may not require use of the entire timer value, and as such the values found at the byte offsets that are frequently changing can likely be ignored. Consider the `SystemTime` value from the Windows `SharedUserData` region of memory. `SystemTime` is a 100 nanosecond timer which is measured from January 1st, 1601, is stored as a _KSYSTEM_TIME structure, and is located at memory address `0x7ffe0014` on all versions of Windows NT[16, 16]:

```
0:000> dt _KSYSTEM_TIME
   +0x000 LowPart          : Uint4B
   +0x004 High1Time        : Int4B
   +0x008 High2Time        : Int4B
```

Due to this timer's frequent update period, granularity, and scale, some of the data contained at the temporal address will be too transient for use as a context-key. The capacity of `SystemTime` is twelve bytes, however due to the four bytes labeled as `High2Time` having an identical value as the four bytes labeled as `High1Time`, only the first eight bytes are relevant as a timer. As shown by the calculations provided by the temporal return addresses paper[16, 10], reproduced below as Figure 1, it is only worth focusing on values beginning at byte index four of the `SystemTime` value, or the four bytes labeled as `High1Time` located at address `0x7ffe0018`.

Byte	Seconds (ext)
0	0 (zero)
1	0 (zero)
2	0 (zero)
3	1 (1 sec)
4	429 (7 mins 9 secs)
5	109951 (1 day 6 hours 32 mins)
6	28147497 (325 days 18 hours)
7	7205759403 (228 years 179 days)

Figure 1: 8 byte 100ns per-byte duration in seconds

It is also interesting to note that if the payload encoder only utilizes a single byte context-key, it may not even be required that the attacker determine the target system's time, as the value at byte index six or seven of the `SystemTime` value could be used requiring only that the attacker guess the system time to within a little less than a year, or to within 228 years, respectively.

3 Weaknesses

Due to the cryptographically weak properties of using functions such as XOR to obfuscate data, there exist well known attacks against these methods and their keying information. Although payload encoders which employ XOR as their obfuscation algorithm have been discussed extensively throughout this paper, it is not the author's intent to tie the the contextual keying technique presented here to such algorithms. Rather, contextual keying could just as readily be used with cryptographically strong encoding algorithms as well. As such, attacks against the encoding algorithm used, or specifically against the XOR algorithm, are outside the scope of this paper and will not be detailed herein.

4 Conclusion

While the use of context-keyed payload encoders likely won't prevent a dedicated forensic analyst from successfully performing an off-line analysis of an exploit's encoded payload, the system it was targeting, and the target application in an attempt to discover the key value used, use of the contextual keying technique will prevent an automated system from decoding the payload in real-time if it does not have access to, or an automated method of constructing, an adequate memory map of the target from which to retrieve the key.

As systems hardware technology and software capability continue to improve, network security and monitoring systems will likely begin to join the few currently existing systems[5, 2–4][4] that attempt to perform this type of real-time analysis of suspected network exploit traffic, and more specifically, exploit payloads.

4.1 Acknowledgments

The Author would like to thank H.D. Moore and Matt Miller a.k.a. skape for their assistance in development of the improved Metasploit implementation of the Shikata ga Nai payload encoder as Proof of Concept as well as

the supporting tools provided by this research effort.

References

[1] Ivan Arce. The shellcode generation. *IEEE Security & Privacy*, 2(5):72–76, 2004.

[2] skape. Implementing a custom x86 encoder. *Uninformed Journal*, 5(3), September 2006.

[3] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, 2004.

[4] Paul Baecher and Markus Koetter. libemu. http://libemu.mwcollect.org/, 2007.

[5] R. Smith, A. Prigden, B. Thomason, and V. Shmatikov. Shellshock: Luring malware into virtual honeypots by emulated response. October 2005.

[6] SkyLined and Pusscat. Alpha2 alphanumeric mixedcase encoder (x86). http://framework.metasploit.com/encoders/view/?refname=x86:alpha_mixed.

[7] SkyLined and Pusscat. Alpha2 alphanumeric unicode mixedcase encoder (x86). http://framework.metasploit.com/encoders/view/?refname=x86:unicode_mixed.

[8] H.D. Moore and spoonm. Call+4 dword xor encoder (x86). http://framework.metasploit.com/encoders/view/?refname=x86:call4_dword_xor.

[9] spoonm. Polymorphic xor additive feedback encoder (x86). http://framework.metasploit.com/encoders/view/?refname=x86:shikata_ga_nai.

[10] vlad902. Single-byte xor countdown encoder (x86). http://framework.metasploit.com/encoders/view/?refname=x86:countdown.

[11] Microsoft. Microsoft security bulletin ms06-040. http://www.microsoft.com/technet/security/bulletin/ms06-040.mspx, August 2006.

[12] I)ruid. smem-map - the static memory mapper. https://sourceforge.net/projects/smem-map/.

[13] Microsoft. Microsoft security bulletin ms04-007. http://www.microsoft.com/technet/security/bulletin/MS04-007.mspx, February 2004.

[14] The Metasploit Staff. *Metasploit 3.0 Developer's Guide*. The Metasploit Project, December 2005.

[15] skape. Safely searching process vritual address space. http://hick.org/code/skape/papers/egghunt-shellcode.pdf, September 2004.

[16] skape. Temporal return addresses. *Uninformed Journal*, 2(2), September 2005.

[17] SweetScape Software. 010 editor. http://www.sweetscape.com/010editor/, 2002.

[18] I)ruid. Memorymap.bt. http://druid.caughq.org/src/MemoryMap.bt, 2007.

A Memory Map File Specification

The memory map files created by this research effort's supporting tools adhere to the file format specification described here. The file format is designed specifically to be simple, light weight, and versatile.

A.1 File Format

An entire memory map file is comprised of individual data records concatenated together. These individual data records represent a chunk of data found in a process's memory space. This simple format allows for multiple memory map files to be further concatenated to produce a single larger memory map file. Individual data records are comprised of the following elements:

Bit-Size	Byte-Order	Element
8	n/a	Data Type
32	big-endian	Base Address
32	big-endian	Size
Size	n/a	Data

Figure 2: Memory Map Data Record Elements

A.2 Data Type Values

The Data Type values are currently defined in the following table:

Value	Type
0	Reserved
1	Static Data
2	Temporal Data
3	Environment Data

Figure 3: Memory Map Data Types

A.3 File Parsing

Parsing of a memory map file is as simple as beginning with the first byte in the file, reading the first three elements of the data record as they are of fixed size, then using the last of those three elements as size indicator to read the final element. If any data remains in the file, there is at least one more data record to be read.

To provide for easy parsing and review of memory map files, an 010 Editor[17] template is provided[18] by this research effort.