Heap Spray Detection with Heap Inspector

Blackhat USA 2011

Aaron LeMasters

8/3/2011

Heap Inspector is a heap visualization and analysis tool. It has the ability to collect a process's heaps using both API and raw methods. Features include searching heaps for string or byte patterns (including regex), dumping heap chunks to a file, and viewing chunks in a hex editor pane. Heaps are displayed visually in a bar chart format known as the heap hash map, allowing the user to view allocations spatially. A similar chart called the heap data map overlays regular expression matches for useful patterns on top of the heap bars. This visualization allows an investigator to quickly discover evidence of a heap spray attack and other useful information stored in an application's heap memory.

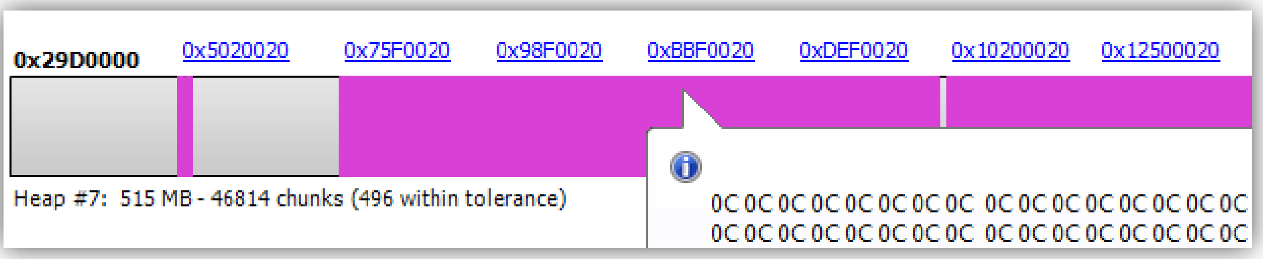
What can I do with Heap Inspector?

In general terms, Heap Inspector allows an investigator to visualize and search data stored in application heap memory. This is a simple yet extremely powerful ability, particularly in the context of host-level forensics. This paper focuses on two specific use cases: detecting heap spray attacks (post-mortem) and searching for personally identifiable information (PII).

Detecting Heap Spray Attacks

A heap spray is an attack technique used to stage shellcode. It is meant to increase the reliability of successfully exploiting a memory corruption vulnerability with an accompanying exploit. This technique is very effective, because it stores the same attack data (which typically contains a nop sled and shellcode) in a large portion of the process's heap memory. Commonly used against applications that host JIT engines (flash, java, etc) such as web browsers and document readers, heap spray attacks are prolific among modern exploits seen in the wild (CVE-2011-0609, CVE-2010-1297, CVE-2010-3973, CVE-2010-3971, just to name a few).

Heap Inspector uses a very simple but powerful concept to make a heap spray stand out in the user interface: the *heap hash map*. This map consists of heap chunks drawn as colored rectangles according to their calculated CRC32, overlaid onto their respective heap. The hash map consists of the CRC32 hash of every heap chunk within tolerance (i.e., greater than a user-supplied base size and frequency of occurrence). Since heap sprays work by allocating the same block of data (a nop sled plus shellcode) hundreds of times on the heap, all of these blocks will have the same hash and will appear as a single contiguous chunk in the UI hash map, as shown below.

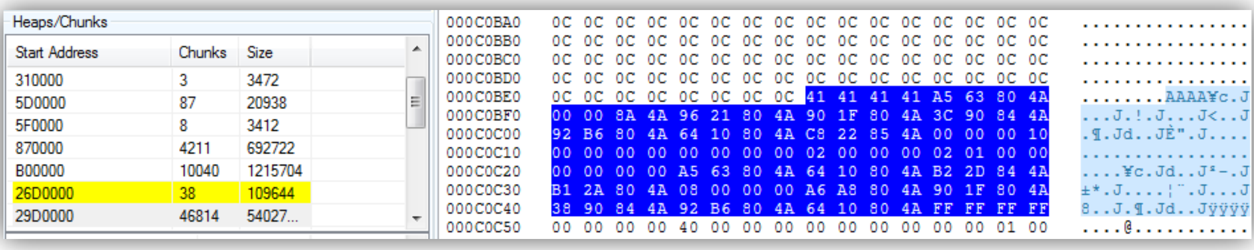


Heap #7: 515 MB - 46814 chunks (496 within tolerance)

```
0C 0C 0C 0C 0C 0C 0C 0C  0C 0C 0C 0C 0C 0C 0C 0C
0C 0C 0C 0C 0C 0C 0C 0C  0C 0C 0C 0C 0C 0C 0C 0C
```

Figure 1: Visualization of successful heap spray in Adobe Reader (CVE-2010-2883)

The nop sled can be seen in the tool tip popup, and the shellcode can easily be extracted by clicking on the chunk address hyperlink, which will switch to the hex editor view pane. Also of note in this screenshot is the heap is abnormally large (515 MB), an obvious indicator of a heap spray.



Heaps/Chunks

Start Address	Chunks	Size
310000	3	3472
5D0000	87	20938
5F0000	8	3412
870000	4211	692722
B00000	10040	1215704
26D0000	38	109644
29D0000	46814	54027...

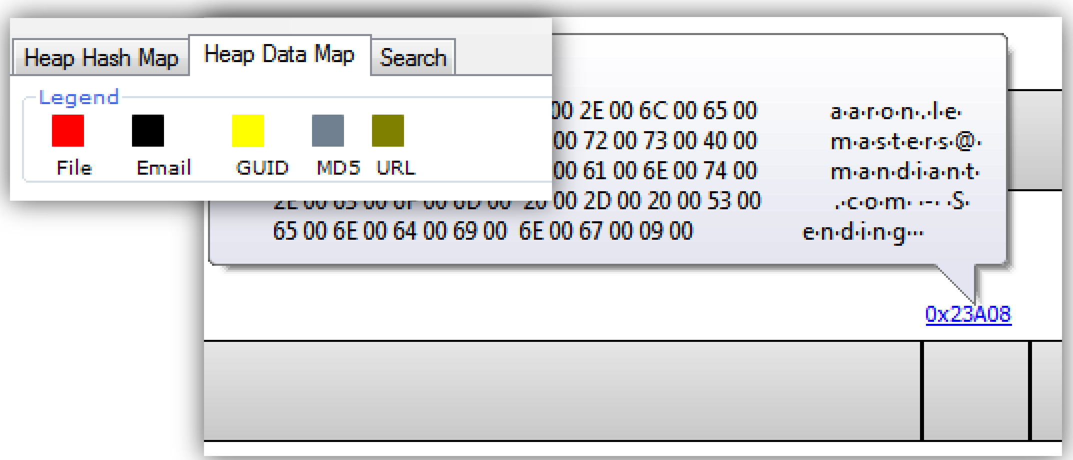
```
000C0BA0   0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C   ................
000C0BB0   0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C   ................
000C0BC0   0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C   ................
000C0BD0   0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C   ................
000C0BE0   0C 0C 0C 0C 0C 0C 0C 0C 41 41 41 41 A5 63 80 4A   ........AAAA¥c.J
000C0BF0   00 00 8A 4A 96 21 80 4A 90 1F 80 4A 3C 90 84 4A   ...J.!.J...J<..J
000C0C00   92 B6 80 4A 64 10 80 4A C8 22 85 4A 00 00 00 10   .¶.Jd..JÈ".J....
000C0C10   00 00 00 00 00 00 00 00 02 00 00 00 02 01 00 00   ................
000C0C20   00 00 00 00 A5 63 80 4A 64 10 80 4A B2 2D 84 4A   ....¥c.Jd..J²-.J
000C0C30   B1 2A 80 4A 08 00 00 00 A6 A8 80 4A 90 1F 80 4A   ±*.J....¦¨.J...J
000C0C40   38 90 84 4A 92 B6 80 4A 64 10 80 4A FF FF FF FF   8..J.¶.Jd..Jÿÿÿÿ
000C0C50   00 00 00 00 40 00 00 00 00 00 00 00 00 00 01 00   ....@...........
```

Figure 2: Extracting the shellcode in heap memory from an exploit of CVE-2010-2883

Once extracted, the shellcode can be analyzed statically in a disassembler or dynamically in an emulator to determine its exact functionality. In summary, the heap hash map makes it trivial to determine the trustworthiness of any suspect document (PDF, MS Word, Excel, etc) simply by running it in a sandbox or virtual machine.

Searching for Personally Identifiable Information

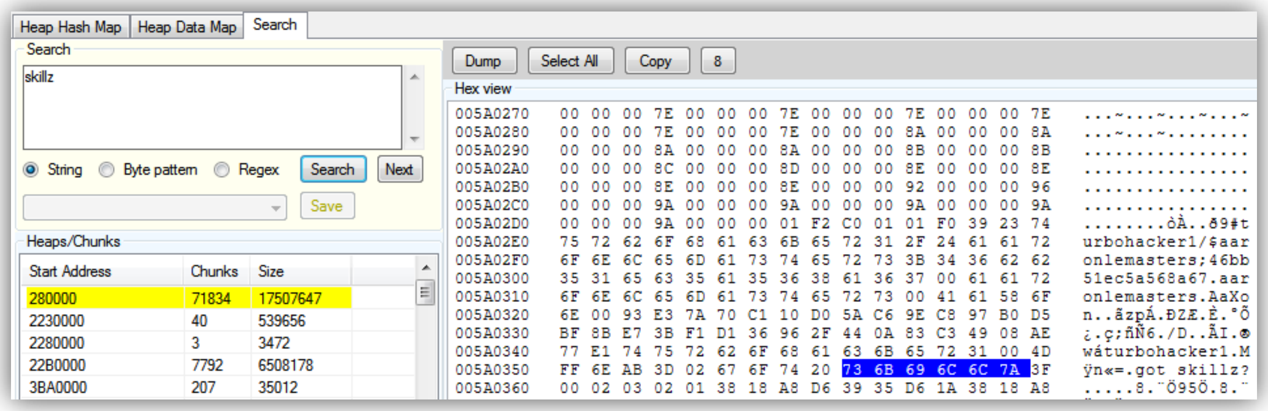
Similar to the heap hash map, the *heap data map* is used to visualize heap allocations, but in this case only heap chunks that match a supplied list of regular expression are drawn on the map. Several default regular expressions are included with Heap Inspector, such as File/UNC Path, GUID, IPv4 address, SQL queries, URL/domains, Social security numbers, and credit card numbers. You can create, import and export your own regular expressions. As shown below, only regular expressions that are matched in heap memory will be displayed.



```
00 2E 00 6C 00 65 00       a·a·r·o·n·.·l·e·
00 72 00 73 00 40 00       m·a·s·t·e·r·s·@·
00 61 00 6E 00 74 00       m·a·n·d·i·a·n·t·
00 2D 00 20 00 53 00       .·c·o·m· ·-· ·S·
65 00 6E 00 64 00 69 00  6E 00 67 00 09 00    e·n·d·i·n·g··
```

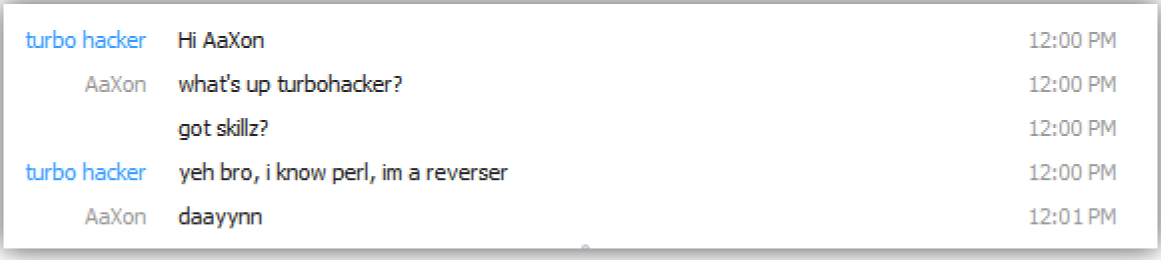
Figure 3: A portion of the heap data map for Microsoft Outlook

In addition to the graphical heap data map, the search tab provides a simple interface to search for regular expressions in application heap memory. The screenshot below shows an active conversation found in Skype's heap memory. There are many other useful items stored in Skype's memory, such as Delphi executable files, SQL statements, and various URLs and configuration data. As a side note, since Heap Inspector reads memory that's in use by the application, anti-debugging and obfuscation tricks do not work (Skype deploys various anti-debugging tricks to prevent attaching with a debugger).



Start Address	Chunks	Size
280000	71834	17507647
2230000	40	539656
2280000	3	3472
22B0000	7792	6508178
3BA0000	207	35012

```
005A0270   00 00 00 7E 00 00 00 7E 00 00 00 7E 00 00 00 7E   ...~...~...~...~
005A0280   00 00 00 7E 00 00 00 7E 00 00 00 8A 00 00 00 8A   ...~...~........
005A0290   00 00 00 8A 00 00 00 8A 00 00 00 8B 00 00 00 8B   ................
005A02A0   00 00 00 8C 00 00 00 8D 00 00 00 8E 00 00 00 8E   ................
005A02B0   00 00 00 8E 00 00 00 8E 00 00 00 92 00 00 00 96   ................
005A02C0   00 00 00 9A 00 00 00 9A 00 00 00 9A 00 00 00 9A   ................
005A02D0   00 00 00 9A 00 00 00 01 F2 C0 01 01 F0 39 23 74   ........òÀ..ð9#t
005A02E0   75 72 62 6F 68 61 63 6B 65 72 31 2F 24 61 61 72   urbohacker1/$aar
005A02F0   6F 6E 6C 65 6D 61 73 74 65 72 73 3B 34 36 62 62   onlemasters;46bb
005A0300   35 31 65 63 35 61 35 36 38 61 36 37 00 61 61 72   51ec5a568a67.aar
005A0310   6F 6E 6C 65 6D 61 73 74 65 72 73 00 41 61 58 6F   onlemasters.AaXo
005A0320   6E 00 93 E3 7A 70 C1 10 D0 5A C6 9E C8 97 B0 D5   n..ãzpÁ.ÐZÆ.È.°Õ
005A0330   BF 8B E7 3B F1 D1 36 96 2F 44 0A 83 C3 49 08 AE   ¿.ç;ñÑ6./D..ÃI.®
005A0340   77 E1 74 75 72 62 6F 68 61 63 6B 65 72 31 00 4D   wáturbohacker1.M
005A0350   FF 6E AB 3D 02 67 6F 74 20 73 6B 69 6C 6C 7A 3F   ÿn«=.got skillz?
005A0360   00 02 03 02 01 38 18 A8 D6 39 35 D6 1A 38 18 A8   .....8.¨Ö95Ö.8.¨
```



turbo hacker	Hi AaXon	12:00 PM
AaXon	what's up turbohacker?	12:00 PM
	got skillz?	12:00 PM
turbo hacker	yeh bro, i know perl, im a reverser	12:00 PM
AaXon	daayynn	12:01 PM

Figure 4: Active Skype conversation discovered by searching for the keyword 'skillz'

The search feature is a powerful tool not only for auditing the data stored and used by an application, but also for reverse engineering internal memory structures. By further analyzing the storage structure of the Skype conversation excerpt above, it could be possible to develop a generic byte signature or regular expression. Such a pattern could then be used for generic Skype conversation detection in subsequent searches with Heap Inspector.

How does Heap Inspector work?

Heap Inspector is a .NET application made up of two primary components:

- **`Heap Inspector.exe`** – main program responsible for most tasks
- **`Be.Windows.Forms.HexBox.dll`** – open source .NET hex editor control

During run time, the main program might extract and use one or more of the following helper programs:

- **`HeapWalker32.dll/HeapWalker64.dll`** – Library injected into target processes to gain access to private process heaps; stored in `C:\Windows\System32\HeapWalker\` and/or `C:\Windows\SysWOW64\HeapWalker\`
- **`Stub32.exe`** – used to obtain required function addresses for WOW64 emulated processes

Heap Inspector works by injecting a "server" DLL (`HeapWalker32.dll` or `HeapWalker64.dll`) into the process you wish to analyze. This DLL sets up a pre-configured, unique named pipe which the main program sends commands to and receives data from. Based on the command it receives, the server DLL uses either Win32 API or raw methods to build a list of available *heaps* and *chunks* (contained in *segments*). The heap chunks contain the actual data we want to collect and analyze. For an in-depth discussion of Windows heap internals, see Chris Valacek's talk [1]. Because of the way Heap Inspector works, it must be executed with Administrator credentials.

Caveats and Technical challenges

Since Heap Inspector relies on DLL injection to collect heap information, it is subject to all of the caveats and technical restrictions inherent to that technique. This includes the possibility of causing instability in the target application as a result of synchronization issues [2]. Such issues arise by virtue of adding a new thread to a remote process that might not be thread-safe. Additionally, accessing and/or locking private heaps might cause deadlock conditions. Furthermore, when the DLL is injected (not using the reflective method), the entry point to each DLL in the process is called. This might result in abnormal or undefined behavior.

There are some tricks application developers can employ to prevent standard DLL injection (e.g., `VirtualAlloc`, `WriteProcessMemory`, `CreateRemoteThread`), such as IAT function hooking and least-privilege concepts (e.g., the process runs with minimal privileges, in a restricted job object, and on its own desktop). Google Chrome uses all of these techniques, making injection difficult. In the case of Google Chrome and other sandboxed processes, even if the DLL is successfully injected using Reflective DLL Injection [3], the injected code can't do anything without first escalating privileges. Due to these obstacles, Heap Inspector is not compatible with Google Chrome or sandboxed processes. Note that it is possible to inject into Chrome's broker process and plugin processes, but this is generally not very useful.

There are other situations where injection might fail, such as across session boundaries in Terminal Services or Vista session separation and in System/service processes. These issues can usually be overcome by injecting as a service or impersonating a user to switch sessions.

When injecting, you must consider the target platforms (32-bit, 64-bit and 32-bit emulated WoW64). There are a few ways to successfully inject across all platforms, but the method used by Heap Inspector is to have an injector stub compiled for each supported platform (32-bit and 64-bit). The injector stub relies on a parameter structure written to the remote process that contains a table of `kernel32` function addresses. For 32-bit and 64-bit native platforms, the .NET application can collect these addresses itself. To support WoW64 processes, the addresses are collected by launching a separate 32-bit application (`Stub32.exe`).

Other miscellaneous challenges include:

- For exception handling in the injected DLL, use OS-provided SEH instead of C++ exception handling, as you have first-chance handling and more fine-grained control over the exception.
- The `smss.exe` process doesn't fully map in `kernel32.dll`, so you cannot inject into this process with Heap Inspector. If you attempted to do so, an access violation would occur, which causes a BSOD.

How is Heap Inspector different than other tools?

There are many ways to collect data stored in memory, such as attaching with a debugger or dumping physical memory using a host of available tools. The most powerful and interactive of these options is a debugger such as `Windbg`, which requires technical expertise and cryptic low-level commands. Physical memory acquisition and analysis is powerful but also requires technical expertise and knowledge of memory structures and operating system internals. It also has the disadvantage of being a "smear" of the contents in memory and instantly stale once acquired.

In general, Heap Inspector seeks a middle ground between technical savvy and functionality. It is as close to "real time" as you can get without using a debugger and does not require the technical savvy to navigate memory contents. Typically in memory analysis, an investigator is dealing with either an entire process dump or an entire physical memory dump, which can be an overwhelming amount of data to cull through. By collecting data only from an application's heap, an investigator can focus on data that's being actively allocated and used by the program.

With regards to string searching functionality, most investigators are familiar with tools such as SysInternals' `strings` utility, which will dump all strings in a binary. Similarly, many memory analysis tools allow an investigator to collect all strings in a process address space, which will include strings from the process's image. In contrast, Heap Inspector only collects strings from the heap, which represents dynamically allocated data used through the life of the program. This distinction is important because it eliminates a large portion of data that might not be useful.

Finally, when applied to heap spray detection, Heap Inspector uses a simple CRC32 frequency-of-occurrence concept to detect and visualize a successful heap spray. All of the current research and tools available for heap spray detection and prevention [5, 6, 7] use a combination of binary instrumentation, emulation, hooking and/or dynamic code analysis to make an informed decision about the contents of heap data. While this has the added benefit of detecting and preventing heap spray attacks in "real time", it has the notable drawbacks of adding overhead and generating false positives/negatives.

Conclusion

Heap Inspector is an easy-to-use, free tool that helps an investigator visualize heap data. Uses of this tool include post-mortem heap spray detection, auditing of personally-identifiable information in application heap data, and reversing of proprietary memory structures stored on the heap.

Future Direction

Currently, Heap Inspector only supports live system analysis by injecting into running processes. This method uses API techniques. In future versions of the tool, raw methods to parse heap structures by using the `PEB` will be employed so that offline memory images can be analyzed.

Also, since the heap spray detection is post-mortem, Heap Inspector will only detect the heap spray after the attack was successful. A future release of the tool might include the capability to detect such an attack in progress. However, several tools and research already exist for this purpose [4, 5, 6, 7, 8] and rely on some variations of binary instrumentation and dynamic code analysis.

Further reading/references

[1] Chris Valasek, *Understanding the Low Fragmentation Heap*, http://illmatics.com/Understanding_the_LFH.pdf

[2] Microsoft, *CreateRemoteThread*, http://msdn.microsoft.com/en-us/library/ms682437%28VS.85%29.aspx

[3] Stephen Fewer, Reflective DLL Injection, http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf

[4] Didier Stevens, *HeapLocker*, http://blog.didierstevens.com/2010/12/06/heaplocker/

[5] Microsoft Research, *Nozzle*, http://research.microsoft.com/apps/pubs/default.aspx?id=76528

[6] Microsoft Research, *Enhanced Mitigation Experience Toolkit v2.0*, http://www.microsoft.com/download/en/details.aspx?id=5419

[7] Akritidis, et al, *STRIDE: Polymorphic Sled Detection Through Instruction Sequence Analysis*, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.5094&rep=rep1&type=pdf

[8] Bania, *JIT Spraying and Mitigations*, http://www.kryptoslogic.com/download/JIT_Mitigations.pdf