Operating Systems Project: Heap Overflow

Authors:
Tysen Radovich (j86j296),
Allen Simpson (r29b821),
Spencer Lawry(w58z387),
Arash Ajam (f63m331)

November, 2020

1 Abstract

Due to an increase in defense against system attacks heap security has been more important. until recently heap overflow attack has been more inconvenient and un-optimized, but with the advances is in other systems in security, Heap overflows have been unaddressed. This has created a surge in the pursuit of finding a way to exploit heap overflows. This has caused a response in bolstering the security of Heaps against such attacks. In this paper, we explain how a Heap Overflow attack works and address the advantages and disadvantages of the four main and one niche solutions to Heap Security. A solution to Heap Security should not make heaps any slower to update and should be able to defend against both malicious and accidental attacks to the heap. That is attacks where the Heap headers are manipulated to point to specific points in memory, and attacks where more data than expected was passed to the Heap buffer causing an overflow of random or generic data.

2 Introduction

There exist unique exploits in each part of a system that aim to take advantage of vulnerabilities and elude security measures. Unfortunately, there is no way to ensure the safety of programs in general; rather, system designers must be aware of each potential weakness that may be targeted. One exception being overlooked is enough to allow someone with malicious intent to break through the defenses. Therefore, it is imperative that operating system designers work to identify areas of weakness and the attacks that will be used to target them to minimize this risk. [11] Many features to protect against attacks come pre-installed on machines these days. [12] The way these features can be used are described in later sections.

As mentioned earlier, specific qualifications must exist for an attack to occur. Most target systems lacking certain types of protection or attempt to disable it, after which a malicious program will have free reign over the system. One common type of attack is called a buffer overflow. There are many different types of buffer overflow, each one attacking a different part of a program or operating system data structure. In this paper, we will focus on the heap buffer overflow attack, which as the name implies, focuses on infiltrating a buffer in the heap section of a program. The purpose of this paper is to explain what a heap buffer overflow is and the circumstances that must exist for it to happen, what an attacker can do if one is successful, and finally, the weaknesses that allow it and how to protect against those. Before jumping into the heap buffer overflow attack though, it helps to understand other buffer overflow attacks for context, which will be covered in the following sections.

3 Buffer Overflow

Buffer overflows are the keys to everything related to every part of creating overflow attacks. A buffer overflow occurs when more data is written to a region of memory than it was originally intended to hold. [11] This can cause issues to arise, or it could be completely benign. In the worst case, these can lead to unintended and unexplainable results in a program. The best case is the program crashes and allows the programmer to identify the bug. A malicious user can exploit a buffer overflow to overwrite some part of memory outside the buffer, thereby corrupting it and potentially giving control to the hacker. For example, by knowing where the return address is on the stack, the attacker can overwrite a buffer-local to a stack frame so that the overflow overwrites the return address with a new address. That new address can point to some malicious block of code so that instead of returning to the location where the program last left off, it jumps to the code of the attacker. From there, the attacker can do just about anything he wants.

Most modern programming languages do not allow copying data into a buffer when the size of that data exceeds the size of the buffer; nor do they allow indexing outside the bounds of the buffer. These checks for buffer size may occur during compile time or runtime. Buffer overflow attacks require that the language/compiler doesn't perform these checks or that they are disabled. If this is true, the attack can proceed by overflowing the buffer by inputting or copying more data than the buffer can hold.

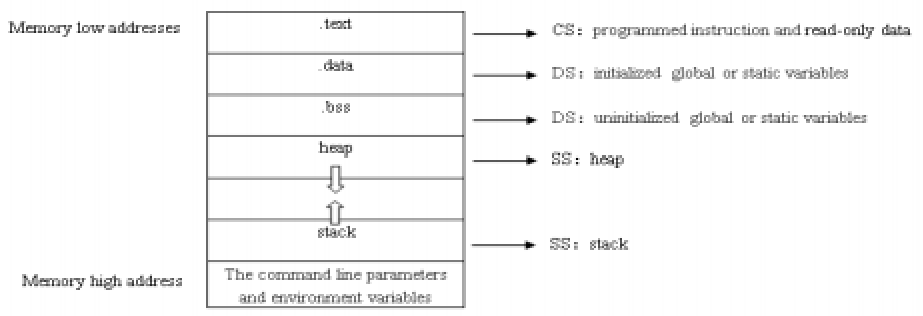

Memory low addresses
.text
.data
.bss
heap
stack
Memory high address
The command line parameters and environment variables
CS: programmed instruction and read-only data
DS: initialized global or static variables
DS: uninitialized global or static variables
SS: heap
SS: stack

Figure 1: Memory layout of a program

A buffer overflow must have corrupted data at some point. By knowing where the the frame pointer(%epb) is I can give it enough position that when the frame pointer is called it goes straight to a malicious piece of code.

4 Stack Buffer Overflows

When one mentions the term “stack overflow”, most computer scientists immediately think of the website that has saved their sanity countless times. Despite that, the name comes from a concept completely separate from that which brings joy and relief. Usually, these occur due to an error in the program which leads to the program’s stack region of memory to exceed the predetermined maximum size of the stack. To be clear, when referring to a stack buffer overflow attack, we don’t mean that the stack itself will be overflowed, but rather a buffer in the stack (such as an array or struct, etc) will be overflowed to exploit the system.

The code below shows an example of a buffer of size 4 called ‘buf’ being overflowed by strcpy with 5 characters, which overwrites the variable ‘a’ to an unintended value, which if it was a password, it would allow an incorrect password to pass the check. This could be abused similarly by a malicious user if the code had a call for input to the user with gets() to fill ‘buf’ rather than this demo example using strcpy(). The excess data in the overflow goes into the region of memory before it on the stack (above if low addresses at top, or below if low addresses at the bottom) because of the first memory location of a buffer the last to be pushed.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void check_user()
{
        char a = 'D';
        char buf[4];

        strcpy(buf, "AAAAA")
        if (a == 'A')
        {
                printf("Correct password\n");
        }
        else
        {
                printf("Wrong password\n");
        }
}

int main()
{

        check_user();
        return 0;
}
```

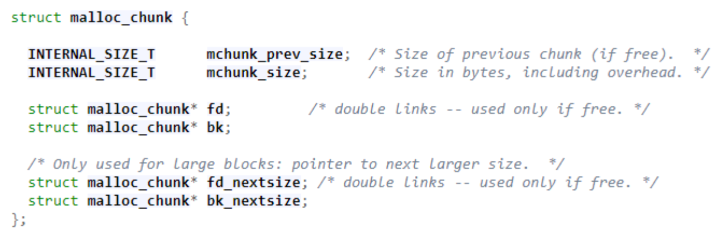
[10]

5 Heap Buffer Overflow

The Heap is an area of memory where data is dynamically allocated. Its size is determined by the heap manager in response to program being run. Since data on the heap can be stored for any length of time it is the heap manager's responsibility to keep a list of the chunks of memory that are free and in use. A Heap Overflow Exploit is only possible if there is a buffer overflow vulnerability due to programming error, typically a missing bounds check. However, Heap Overflows are significantly more difficult to exploit than stack overflows because the structure of the heap can be architecture specific.

Recall that in a Common Heap structure there are 2 main parts, the Header Block, and the Free List. The Header Block sits above the data block in the Heap and contains the metadata of the data structure, such as size, free or in-use state, state flags for the heap manager, and the data section for user data. If the chunk is free, the data section instead includes the forward and backward pointers to other free chunks, as consecutive blocks of data in the heap needn't be next to each other in

memory. The Free List is a list of all the chunks in the Heap that are currently free and is used by the Heap Manager to search for valid locations in the heap to allocate data. A heap buffer is overflowed when more data is supplied to the data block than the block is actually allocated for. This means that the next block of memory's header information is overwritten. When that header is overwritten, it could lead to gibberish being written into the header which could cause a crash, or it could be used as a door for an attacker to change the forward and backward pointers to point to a memory location of an attacker's supplied data leading the next time that memory block being called to execute code supplied by the attacker. Dongfang Li explains how the process works succinctly,
[8]"Heap overflow occurs when a program writes data to a buffer, then overruns the buffer's boundary and overwrites adjacent memory. Attackers can exploit heap overflow vulnerability to cover critical data, such as function pointers, heap management structure, and other useful data. Then programs fail to run, or perform other code which was pre-injected into buffers."



```
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

Figure 2: Heap Header Structure

6 Security Measures

There are many countermeasures that can be taken to prevent Heave overflows such as to add *Canaries*. Canaries, just like their real-life counterparts are used to check if the current environment is not safe. More clearly, a Canary is a data value set to a specific value or a value that can be validated by the heap manager as valid before the heap manager tries to allocate, deallocate, or access data on the heap. This means that, should a Heap Overflow attack override the header, it will likely set the value of the canary to something that would not be valid. This is an effective solution against accidental Heap overflows as the likelihood of random data overwriting the header putting a valid value in the canary is low. However, against a malicious attack, it is possible for the canary to be avoided by tailoring the injected code to keep the canary value valid. Unfortunately, canaries require significant overhead as the canaries must be checked on every heap update (`malloc()` and `free()`) and you would have at least as many canaries as Heap data blocks. This means the already slower speed of the heap becomes an even more pronounced issue.

Another method of protecting against heap overflows is to use *Guard Pages*. Guard Pages are pages or blocks in memory that remove the write permissions. If Guard Pages are placed in front of each block of data in the heap they prevent overflows from altering the data in adjacent blocks. Guard pages have their advantages and disadvantages. On one hand, Guard pages offer a good solution with a small amount of code and offer that security without non-error states being much more expensive. But on the other hand, Guard pages are architecture-specific and require additional effort to program correctly. Guard pages also become very expensive when an error state occurs as permissions must be verified. Additionally, Guard pages can make debugging the heap finicky at best and hellish at worse as messing with segmentation faults tends to cause issues during debugging code. [9]

Another solution is to store the data and heap randomly. This requires the Heap manager to additionally choose a random location in memory to store the data each time it is requested to allocate and to only reuse blocks of memory and free lists randomly. This protects against malicious Heap Buffer Overflows from targeting specific blocks or knowing with any certainty what block of data is free and will be adjacent to the block that they are overwriting making a Heap Overflow Attack much more difficult. This solution does not protect from Accidental Heap Overflows very well as headers can still be overwritten, however it protects well against attackers.

A particularly novel approach to protecting against Heap Overflow is to not store the Heap Metadata Inline, but rather to store *all* the Heap metadata at either the beginning or end of the Heap and surround it with Guard pages. This means that Heap overflow attacks cannot change the pointers in the headers and that you can get away with *significantly* fewer guard pages. This solution much like the previous option is very effective against targeted malicious attacks as it prevents an attacker from injecting code. This solution, like the previous solution, does not protect against Accidental Heap Buffer Overflows, as, while messing up the headers is now out of the question, any Overflow will instead overwrite the data in the next block. While this will not *likely* cause a crash, it will more than likely corrupt data which may be worse for the user.

The more advanced the solution is to solve a heap overflow the more niche it becomes. While these solutions may be more niche, they are also more effective at addressing the issues that show up in the previously mentioned solutions.

The HeapDefender is a unique way to help protect against heap overflow attacks.

According to Li, "HeapDefender is a fine-grained instruction stream monitoring hardware defense mechanisms. We extract the pre-defined heap memory range and then check whether the address of heap operations of instructions exceeds the heap address range in the running time of embedded processors."This means that this is constantly checking to see if each instruction is bigger than what it was supposed to originally contain. This is then handled by 4 components. The first one being the function recognition unit. This will take the instruction that it was given and check if it is safe. If it is safe it sends it to two other blocks: the heap range storage unit and the heap operation address unit. The heap range storage unit takes in the function recognition unit from the previous function. The second place the function recognition unit sends the signal to the heap operation address unit which determines what that memory range does. Then the comparator unit takes the signals from the heap range storage unit, the heap operation address unit, and sees if the range in memory correlates with what is supposed to be located there. If they do not correlate and are not the same size then it sends an alarm that a heap buffer overflow attack is happening.[8]

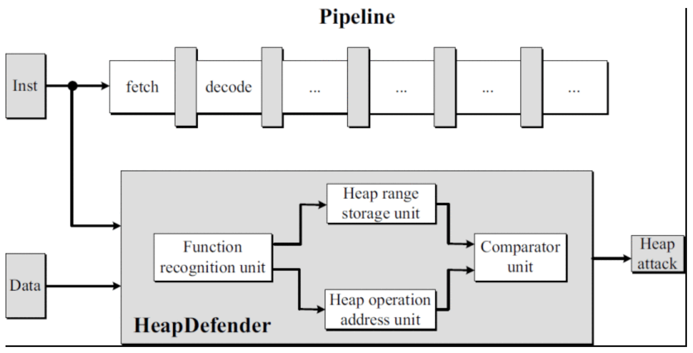


Figure 3: HeapDefender

This diagram shows the way that it works alongside the CPU to show that it is working.

7 Conclusion

Heap buffer overflow attacks prove to be a complex issue and while solutions exist to protect from them, no single solution is a perfect catch-all to solve the risk of all forms of these attacks. The various countermeasures presented in this paper give potential solutions to them, but as with everything, they come with trade-offs in performance (speed/space) and/or simplicity of use. While a full-proof solution might be to use languages that perform bounds checks at compilation and during runtime and don't allow direct memory access (like pointers), this isn't practical because C and C++ are still used in practice. Heap buffer overflow attacks historically have been less common than the stack equivalents, and are recently becoming more prevalent, leading to more the need for new solutions to be developed that were unnecessary until now. Through continued research in this topic, more effective and general solutions may be found to this ever-elusive conundrum that is ***Heap Buffer Overflows***.

References

[1] Black, Paul E, and Bojanova, Irena. “Defeating Buffer Overflow: A Trivial but Dangerous Bug.” IT Professional, vol. 18, no. 6, 2016, pp. 58–61.Web.9 Nov. 2020.
`https://ieeexplore-ieee-org.proxybz.lib.montana.edu/document/7763738`

[2] Dalton, Michael, et al. “Real-World Buffer Overflow Protectio n for Userspace &amp; Kernelspace.”USENIX,USENIX.Web.9 Nov. 2020.
`www.usenix.org/legacy/events/sec08/tech/full_papers/dalton/dalton_html/`

[3] Dongfang Li, et al. “HeapDefender: A Mechanism of Defending Embedded Systems against Heap Overflow via Hardware.” 2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing, 2012, pp. 851–856.Web.9 Nov. 2020.
`https://ieeexplore-ieee-org.proxybz.lib.montana.edu/document/6332095`

[4] Du,Wenliang.”COMPUTER & INTERNET SECURITY:A Hands-on Approach 2nd Edition”.Syracuse University,Wenliang Du, May. 2019.

[5] Foster,James C.,et al. ”Buffer Overflow Attacks Detect”, Exploit, Prevent, 2005.Web.9 Nov. 2020.
`https://ebookcentral.proquest.com/lib/montana/reader.action?docID=222803` .

[6] Huang, Ning, et al. “Analysis to Heap Overflow Exploit in Linux with Symbolic Execution.” IOP Conference Series. Earth and Environmental Science, vol. 252, 2019, p.Web.9 Nov. 2020. 42100.
`https://iopscience-iop-org.proxybz.lib.montana.edu/article/10.1088/1755-1315/252/4/042100`

[7] R. Hund, C. Willems and T. Holz, ”Practical Timing Side Channel Attacks against Kernel Space ASLR,” 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, 2013, pp. 191-205, doi: 10.1109/SP.2013.23.Web.9 Nov. 2020. `https://ieeexplore.ieee.org/abstract/document/6547110`

[8] Li, Dongfang, et al. “HeapDefender: A Mechanism of Defending Embedded Systems against Heap Overflow via Hardware.” IEEEXplore, IEEE, 18 Oct. 2012,Web. 9 Nov. 2020
`ieeexplore-ieee-org.proxybz.lib.montana.edu/document/6332095` `https://www.infona.pl/resource/bwmeta1.element.ieee-art-000006332095`

[9] Marlow, Simon and Jones, Simon Peyton ”Multicore Garbage Collection with Local Heaps” Microsoft Research, Cambridge, U.K.Web.14 Nov. 2020. `http://simonmar.github.io/bib/papers/local-gc.pdf`

[10] Thakur, Abhishek. “Stack Overflow Vulnerability.” StackOverflow Vulnerability, Hacker Noon, 7 Jan. 2020. Web 9 Nov. 2020 hackernoon.com/stack-overflow-vulnerability-xou2bbm

[11] Xu, Shu Xin,and Chen,Jun Zhang. “Analysis of Buffer Overflow Exploits and Prevention Strategies.” Applied Mechanics and Materials, vol. 513-517, 2014, pp. 1701–1704.Web.9 Nov. 2020. https://search-proquest-com.proxybz.lib.montana.edu/docview/1718987817?rfr_id=info%3Axri%2Fsid%3Aprimo

[12] Zhang, Chao, et al. “Using Type Analysis in Compiler to Mitigate Integer-Overflow-to-Buffer-Overflow Threat.” Journal of Computer Security, vol. 19, no. 6, 2011, pp. 1083–1107.Web.9 Nov. 2020. http://web.a.ebscohost.com.proxybz.lib.montana.edu/ehost/pdfviewer/pdfviewer?vid=1&sid=ef60d671-ee06-4cc4-b366-5681dbab5cea%40sessionmgr4007