2016 49th Hawaii International Conference on System Sciences

Mitigating 0-days through Heap Techniques - An Empirical Study

Lucas McDaniel
University of Alaska Fairbanks
lamcdaniel@alaska.edu

Kara Nance
University of Alaska Fairbanks
klnance@alaska.edu

Abstract

Securing public-facing services is a challenging task for all types of users and even best practices might not be sufficient at stopping attackers with an 0-day. It is often the case that when a new vulnerability is discovered, there is a race between attackers to exploit the vulnerability, and system administrators to patch the system in a manner that does not break existing functionality nor induce an unnecessary amount of downtime. For individuals hosting publicly accessible services such as a website or data storage, this race greatly favors the attacker as the average system administrator may not be aware of the need to update a system until days or even weeks after proof-of-concept exploits are made publicly available.

In this paper we conduct exploratory research into techniques that can hinder the successful exploitation of a service with minimal impact on the system. While these techniques do not prevent a 0-day from exploiting the service in all cases, it can be used to defend a system against some types of bugs and against some types of techniques used by exploit developers. The ultimate goal of this research is to present methods that may prove to be successful at buying an individual time before it is necessary to patch the system by bypassing the initial round of exploits. We conclude this paper by demonstrating the potential of these methods on several modern systems including some with publicly available exploits.

Keywords: Attack mitigation, attack prevention, computer security.

1. Introduction

Securing computer systems from attackers is an increasingly challenging task for not only large corporations, but also individuals. In recent years, it has become remarkably simple for users without any technological background or understanding, to set up systems that were previously inaccessible to them. With only a few clicks on Amazon's AWS Marketplace [1], users can deploy their very own VMs serving as blog, email stack, VPN server, or even a cloud storage node without ever considering any technical details or security issues that may arise. The result is that many of the users are unaware of proper techniques to secure their environments, methods to detect a successful attack, and how to recover after such an attack.

While reducing the burden required to setup common types of services through the use of existing software is convenient, it also comes with intrinsic weaknesses: untold numbers of homogeneous systems. A single vulnerability in a common service can now affect a myriad of users, such as the recent Drupal SQL injection vulnerability that affected approximately 12 million sites [2]. This environment is a dream for an attacker with one such exploit as the exploit can be used with a high-degree of reliability across all vulnerable sites. Even better still, the users in charge of the maintenance of these sites will often not patch their systems in a reasonable timeframe thus giving the attacker ample opportunities for success.

Having a large number of homogeneous systems isn't inherently bad; it simply means that when something goes wrong an attacker has the advantage, especially over users that are not security-conscious. However, it also means that there are known lists of minor tasks that individuals need to do in order to secure their environments such as setting good passwords, keeping packages updated, using encrypted channels when accessing administrative capabilities, etc. Provided users follow the best practices for securing their systems, they will be awarded with "good-enough" security.

In this context, good-enough security is a loosely defined concept that means the system is secure against most common types of attacks. Unlike corporations and governments that are actively concerned that they are the targets of an Advanced Persistent Threat (APT), individuals can get away with less-stringent security requirements placed on their systems. Their primary security concern is in ensuring (or, rather, hoping) that their services will get passed over by an attacker who

1530-1605/16 $31.00 © 2016 IEEE
DOI 10.1109/HICSS.2016.689

has access to an 0-day as they don't need or require perfect security for their day-to-day operations.

In this paper, we present a simple technique that can offer such a result against a class of vulnerabilities and exploits. Just as changing user credentials and updating packages in a timely manner are widely accepted to be successful at preventing a variety of attacks, we propose minor modifications to system libraries that result in an altering the heap layout can also offer significant preventative capabilities at little cost. These changes aren't designed to anticipate and prevent an exploit from happening, but instead reduce the effectiveness of an attack in hopes that the attacker will simply pass over the service instead of pursuing it further.

In the following sections of this paper, we will provide an overview of heap memory, discuss the history of heap overflows and modern techniques, motivate how modifications to the heap layout can prevent successful exploitation, and conclude with empirical analysis of its effectiveness at hindering recent exploits.

2. Heap Memory Overview

Before digging too deeply into these topics, it is useful to review at a high-level what heap memory is, how it's different from other types of memory (e.g., the stack), and its standard usage within C-style languages. Loosely put, the heap contains all memory whose size is not known at compile time. Any time a program needs to handle strings that can be of arbitrary size or lists with arbitrary numbers of elements, it is likely that this data is stored in the heap. The heap also contains data that needs to be stored and referenced at a later time as once data is created in the heap it will exist until the space is freed. This also means that the software developer is responsible for requesting and freeing memory, as well as ensuring appropriate operations are done on this memory as this is not automatically managed for them. If done improperly, this can be the source of software vulnerabilities.

This differs from the stack where the space for the data is only allocated until the function that created it returns, which makes it ideally suited for storing local or temporary values. Because the amount of stack memory needed by each function is known at compile time, the layout of the stack is predictable at any point during the execution as shown in **Figure 1**. In this figure, the buffers are local data and the function header includes function parameters, saved stack pointer, etc. The image shows the initial stack layout in Function 1, the layout when Function 2 is called, and finally the layout when Function 2 returns. This figure demonstrates that offsets between values stored on the stack can be determined in advance, or at least exist within some guessable/predictable range. The stack is also used to store register values, pass parameters, store return addresses, and more, depending on the calling convention being used.

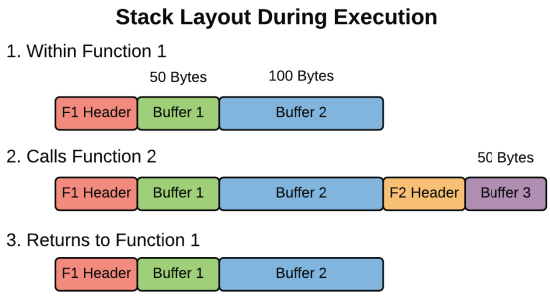

***Figure 1**: Simplified stack layout.*

While the layout of the stack can generally be determined by simply looking at the current stack trace, the layout of the heap cannot. As memory is requested and freed, empty blocks of memory can be reused if they are larger than the newly requested size. Depending on the flow of execution and the size of the data requested during execution, the heap can change significantly as shown in **Figure 2**. The buffers are heap allocated memory with the first step showing the layout prior to a variable sized buffer allocation. The second step shows the resulting layout if the new buffer is 50 bytes, and the third step shows the layout if instead the new buffer is 150 bytes. It is clear that the heap structure is dependent on the buffer size, which is not generally predictable.

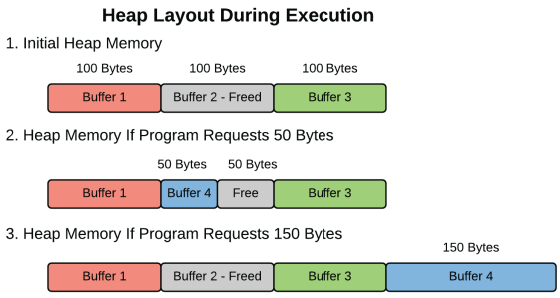

***Figure 2**: Simplified heap layout.*

3. History of Heap Memory Exploitation

Not all software vulnerabilities are made equally, and even seemingly identical vulnerabilities may not all be exploitable when the context of the rest of the system is considered. Exploit development largely

focuses around finding commonalities within vulnerabilities (i.e., to form classes of vulnerabilities) with the ultimate goal of finding out how those with these commonalities can be reliably exploited. By identifying the minimum requirements that must be met by the vulnerability, exploitation techniques can be developed independently from the vulnerability and research into developing new techniques can be conducted without having any specific target or usage in mind. Occasionally, new techniques are created that allow for the successful exploitation of a class of vulnerabilities that were previously not exploitable [3].

In the 1990s and earlier, stack smashing was one such commonly used technique. The ingredients for successful exploitation were simply a buffer overflow on the stack of sufficient size and controllable data. This allowed an attacker to overwrite the current return address to call shell code that was also stored on the stack. Shortly after the seminal publication of *Smashing The Stack For Fun and Profit* – an article on the cutting edge of research for such techniques at the time – in 1996 [4], modifications were made to compilers to prevent the effectiveness of these exploitation techniques [5]. While the techniques described in that paper are no longer useful in exploiting most modern systems, it still serves as a practical example of how analyzing the necessary requirements for exploitability can be done independent of the vulnerability being exploited. This trend for research is still very prominent even today.

These new stack buffer overflow detection and prevention techniques forced exploit developers to identify new methods for exploiting a vulnerability. One common venue for attack was the heap memory allocator used by *glibc* called *dlmalloc* or Doug Lea Malloc after its primary developer [6]. By storing memory management information in-band (i.e., within the heap next to the memory it is managing), *dlmalloc* was ideally suited for portability purposes but also for exploitation. This meant that any heap buffer overflow of sufficient size and controllable data would be able to overwrite part of the *malloc* chunk structure, which would be exploitable through techniques described in articles such as *Vudo malloc tricks* [7], *Once upon a free()* [8], and *Advanced Doug Lea's malloc exploits* [9].

As with before, once tricks and techniques became available to ease exploitation, defensive countermeasures were introduced to limit their effects. Addressing existing limitations in *dlmalloc*'s ability to verify the link chain in the *malloc* chunks was sufficient to prevent successful execution of many of the techniques described in the previous articles. Another article was released some time after titled *Malloc Maleficarum* [10], which sought to find weaknesses in these prevention methods, but as the name may imply these new techniques were largely esoteric and didn't gain the wide spread adoption that previous efforts had received.

4. Current Techniques

This is not to say that modern techniques no longer build on the work done by previous exploit developer research, nor that esoteric techniques are not important to modern applications. A recent publication titled *Malloc Des-Maleficarum* [11] took the theoretical nature of its predecessor and offered practical examples for its use. Similarly, a recent article by Google's Project Zero demonstrated how a single null byte overwrite could lead to privilege escalation on a target system [12]. However, while there is certainly still work being done to observe weaknesses in heap memory structure protections – just as there is still work done to find similar weaknesses in stack buffer overflow detections – common areas of research focus on abusing other aspects of an executable.

Recent work has demonstrated the effectiveness of code-reuse style attacks against a program of sufficient size. Given the attacker' ability to alter the flow of control through a single jump, call, or return instruction, it is possible that he can execute a section of existing code with different parameters that will perform a malicious action. Given an overflow of some kind, this operation is possible by overwriting a function pointer stored somewhere in writeable memory, so that when the program attempts to call the intended function, it instead calls the attacker-controlled address. The full exploit is normally not done through a single call, but rather a single call to a specific section of code – referred to as a stack pivot for Return-Oriented Programming (ROP) or dispatcher gadget for Jump-Oriented Programming (JOP) – that enables sequences of other calls to be performed in an attacker controlled method [13] [14]. When the programming is utilizing *libc*, this style of technique has even been shown to be Turing complete [15]!

5. Breaking the Exploit

The goal of this paper and the research project it is covering is to demonstrate the effectiveness of a minor modification to *glibc* at hardening a system against an exploit. While the exploit techniques being targeted with this effort are those based on heap overflows that primarily lead towards ROP and JOP capabilities, the same modification may also affect other styles of attacks as well. It should be noted that this effort would NOT remove a vulnerability from a system nor

make an exploit against it impossible. Instead the goal is to identify common assumptions made by exploit developers, and create modifications that invalidate these assumptions.

When building an exploit, it is common to have a specific target system in mind and build the attack accordingly. In some cases this is done because the vulnerability only exists on a very specific type of system, or because the exploit may need to be altered in significant ways given the characteristics of a specific target. For instance, minor changes to the source code can result in minor changes to the offsets of functions in the resulting binary. Initially, an attacker might identify an appropriate set of commands for a stack pivot or a dispatcher gadget. He will then determine what offset in the binary these commands are located at given the version of the binary on the target system. An incorrect guess here may result in no discernable effect on the system, or may result in a crash instead of the intended arbitrary code execution. Offsets, relative locations, and base addresses of the executable, shared libraries, heap, and the stack are all values that may need to be known or computed depending on the nature of the bug and the exploitation technique being used against it.

Offsets for the location of functions in both the executable and shared libraries are hard to alter dynamically, but other offsets such as those in the heap are easier. Given an overflow that allows for both reads and writes of data, it is common to attempt to read pointers to the heap, stack or functions that can be found at predetermined offsets in order to bypass address randomization. However, when slight changes are made to these offsets, it is possible that the exploit code will be written in such a way that it is unable to find the values necessary to bypass address randomization even if it is still theoretically possible for them to do so. By making minor changes to the heap layout, we can effectively invalidate many of the assumptions made by exploit developers regarding the locations of needed information and potentially weaken the effectiveness of the exploit code.

For this exploratory research, we have chosen to make the following single line modification to *gibc*:

```
2908 void*
2909 public_mALLOc(size_t bytes)
2910 {
2911   mstate ar_ptr;
2912   void *victim;
2913
2914     __malloc_ptr_t  (*hook)  (size_t,
__const __malloc_ptr_t)
2915     = force_reg (__malloc_hook);
```

Excerpt from unmodified malloc/malloc.c in version 2.15 of glibc (Ubuntu 12.04 default).

```
2908 void*
2909 public_mALLOc(size_t bytes)
2910 {
2911   mstate ar_ptr;
2912   void *victim;
2913   bytes += 16;
2914     __malloc_ptr_t  (*hook)  (size_t,
__const __malloc_ptr_t)
2915     = force_reg (__malloc_hook);
```

Excerpt from modified malloc/malloc.c in version 2.15 of glibc (Ubuntu 12.04 default).

This single line change means that all malloc requests are increased by 16 bytes and then processed as normal. By making this simple modification, the relative layout and offsets of data in the heap are altered in ways that may not be anticipated by exploit developers and potentially reducing what would normally be remote code execution attacks into denial of service attacks through crashes in services.

This modification will change the layout of memory in two ways: small spaces between consecutive structures in memory, and changes in relative layout as show in **Figures 3** and **4**. The first step in figure 3 shows the initial layout of heap memory, which consists of 4 buffers each of 50 bytes. Step 2 involves the middle buffers – Buffer 2 and Buffer 3 – being freed under standard operation. Step 3 has the program requesting a new buffer of 110 bytes. Because this buffer is larger than the previous freed area, it must be *alloc*'d somewhere else.

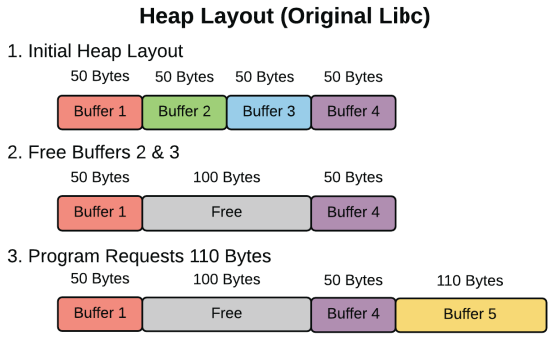

***Figure 3**: Simplified heap layout running the unmodified version of libc.*

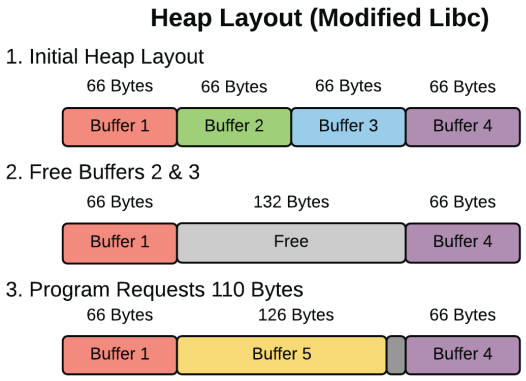

***Figure 4**: Simplified heap layout running the modified version of libc.*

In figure 4, the first step shows the initial layout of heap memory, which consists of 4 buffers each of 50 bytes. Since the modifications made to *libc* will append 16 bytes onto each buffer, their resulting size is 66 bytes. Step 2 involves the middle buffers – Buffer 2 and Buffer 3 – being freed under standard operation, which now creates 132 bytes of free space. Step 3 has the program requesting a new buffer of 110 bytes. This request will result in 126 bytes being *alloc*'d in the previous free space with a very small section of free space remaining after. In this example, Buffer 4 and Buffer 5 are no longer in the same relative order as they were under execution of the unmodified version of *libc.*

As shown above offsets between neighboring structures will increase by 16 bytes making exploits with predetermined hardcoded offsets ineffective. Given standard operations of *malloc*ing and freeing data, it is possible that relative layouts are also changed given free space is no longer large enough to fit newly requested data within, as shown in the above figures. This means that if a certain buffer in the heap could only be overflowed in one direction (e.g, Buffer 4 expects Buffer 5 to be immediately after it, as it can only overflow in that direction), it is possible that the vulnerability is no longer exploitable at least in the same general fashion. Note that in both of these images, *malloc* header data is not shown in order to simplify the image.

6. Empirical Results

In order to test the efficacy of this technique at altering heap structures, we have performed various comparisons between the execution of the program with unmodified and modified libraries. The first set of tests consisted of running several common commands to determine changes within the heap between executions. The second set of tests involved setting up vulnerable versions of applications and testing public exploits against these applications first with the original version of *glibc* and then with the modified version replaced system wide.

6.1 Evaluating Heap Changes

A script was written that runs a program with both versions of *glibc* in order to log changes in the heap. The different versions of *glibc* were loaded using LD_PRELOAD and a second wrapper library was also loaded at this time to log calls to *malloc* and *free*. This wrapper library was used to determine the state of the heap at each point during execution. While this is not a comprehensive method of identifying the allocation of memory within the heap, it serves as an appropriate approach for the tests. Each program was executed multiple times to ensure that ASLR was disabled and execution was deterministic.

The heap state after the last *malloc* during program execution was used to compare the differences between the libraries. This analysis relied on the fact that the programs being tested were deterministic in nature so that the orders of *malloc* calls could be used to uniquely identify identical buffers in memory (i.e., the first call to *malloc* for both version of the library create buffers for the same logical variable, the second also creates the next logical variable and so forth). This provides the means for analyzing differences in the heap by tracking the order of calls.

Three statistics were recorded for each program execution: average change in heap offset from the start of the heap, standard deviation of these offsets, and percentage of buffers no longer in relative order. The changes in heap offset were determined by calculating the difference between the given buffer and the first buffer *malloc*'d. Since it is common for exploits to identify the base of a memory region and calculate offsets from that base, this is indicative of how far off hardcoded values might be, on average. The third statistic served as an estimate for the number of buffers that were out of place. This was calculated by taking a given buffer and counting the number of other buffers that came before (or after) it in memory when running the unmodified *glibc*, but now came after (or before) it when running the modified *glibc*. These numbers were averaged across all buffers after eliminating cases of double counting. Table 1 shows these statistics for a selection of common bash utilities.

Figures 5 – 12 show the heap layout of the same program when ran with both versions of *glibc*. The top level is the heap with the unmodified version of *glibc*, and the bottom level is with the modified version.

Program Name	Buffers in Heap	Average Offset	Standard Deviation	Positional Changes
/usr/bin/stat	152	1270.	1531.11	0.64%
/bin/df	165	1406.0	2292.14	2.35%
/usr/bin/w	109	2464.0	17317.40	1.07%
/usr/bin/find	82	1046.0	1793.75	0.51%
/bin/ls	60	456.0	6030.25	0.12%
/sbin/fdisk	66	665.0	968.61	0.21%
/bin/uname	31	240.0	279.43	0.00%
/usr/bin/nm	14	442.0	1231.29	0.07%

***Table 1**: This table shows statistics regarding the differences between the heaps immediately after the last malloc.*

***Figure 5**: Heap memory of /usr/bin/stat*

***Figure 6**: Heap memory of /bin/df*

***Figure 7**: Heap memory of /usr/bin/w*

***Figure 8**: Heap memory of /usr/bin/find*

***Figure 9**: Heap memory of /bin/ls*

***Figure 11**: Heap memory of /bin/uname*

***Figure 10**: Heap memory of /sbin/fdisk*

***Figure 12**: Heap memory of /usr/bin/nm*

Buffers are colored on a uniform spectrum starting from red, which means that buffers in the red spectrum are allocated first and the buffers in the purple-red spectrum are allocated last. Buffers that are the same color represent the same logical variable in memory, and white spaces between buffers represent unused space or *malloc* header data. The width of the buffers and whitespace is the logarithm of the buffer size.

Observing **Figure 13** – the memory layout for /usr/bin/nm – we see that buffers are the same size with slight increases in the spacing between buffers. This is due to the fact that we are adding 16 bytes to each buffer thereby increasing the size of unused regions in the heap. However, under standard operations of *mallocs* and *frees*, we see that differing amounts of space was empty between executions thereby allowing the buffers to be allocated at different addresses.

***Figure 13**: Zoomed heap memory of /usr/bin/nm*

This confirms the theoretical notion that reordering of buffers is possible through this modification to *glibc*.

6.2 Evaluating Exploits

Several exploits were tested against vulnerable applications to determine if the modifications to *glibc* prevented the exploit from succeeding. For each case, the system was confirmed to be vulnerable by a common public exploit prior to *glibc* being replaced system-wide. While the number of exploits tested is far from a comprehensive list, we include a brief analysis of some high profile exploits to demonstrate the efficacy against modern attacks.

CVE-2015-0240 is vulnerability against Samba that allows for unauthenticated remote root. This vulnerability was first reported February 23, 2015 with public exploits available as early as February 28, 2015. Given the pervasiveness of the service, the criticality of the vulnerability, and the nature of the vulnerability – freeing uninitialized and controllable heap memory – it was ideally suited to test the exploit against the modified *glibc*.

Several different proof of concept (POC) of the exploit were tested against Samba running on Ubuntu 12.04, and none were still able to successfully gain root access against the target after replacing *glibc*. (In fact, none of the examples tested appear to have caused any adverse affects to the system such as crashes or instability although this aspect was not thoroughly tested.) One of the better-documented POCs was analyzed, and it appears to have failed due to a logic error when the exploit is unable to find a specific buffer in the heap in order to identify the heap's base address (line 392) [16]. This means that, had a vulnerable Samba server been using the modified *glibc*, while it may have still been exploitable, none of the tested POCs would have succeeded.

CVE-2014-5119 is vulnerability against *glibc* popularized by Google's Project Zero. First reported on July 14, 2014, it was not until August 25, 2014 that a public exploit was released showcasing a local privilege escalation through a poisoned NUL byte in *pkexec*. Again, given the high-profile nature of this bug, it was also ideally suited for testing.

The most recent version of Google's POC was tested against *pkexec* running on a Fedora 20 32-bit desktop. Minor changes to the POC's profiling of the memory locations were necessary to get it working against the unmodified version of *glibc*. Replacing *glibc* system-wide resulted in the exploit being unable to gain privilege escalation as the POC – which is based heavily upon spraying the heap in order to get it to collide with the stack – requested too much memory and failed to execute (see: pty.c, line 152) [17]. Again, this type of modification to *glibc* was able to prevent the successful exploitation of the system.

6.3 Analysis

It appears that there is a correlation between increases in buffer size in the heap, and increases in changes to relative layout and relative offsets. Visualizations of these results can be seen in **Figures 5-12.** These results make sense and are expected as chaos theory states that small changes in one aspect of a system can produce significant changes over time. So, it is anticipated that minor changes to the heap layout and offsets with each *malloc* will eventually combine to form significantly larger changes. Although the analysis done was limited to small programs, these preliminary results indicate that larger and more complex programs would have larger and more drastic alterations to their heaps. This was supported by our analysis of the efficacy of exploits against vulnerable applications after changing the version of *glibc* system-wide.

While the modifications were consistently effective at preventing successful exploitation in the test cases, it is worth mentioning that the vulnerabilities chosen were handpicked to ensure a high chance of success. There is no indication that these results can be extended to all cases. For example, SQL injections would likely be unaffected since they do not require modifications to operations as the binary level. Similarly, any denial-of-service vulnerability would largely be unaffected as the intended best-case scenario of this modification is to cause a crash instead of the successful exploitation of a system. However, the empirical results show that this style of modification is promising for future research especially when considered in the context of this type of vulnerabilities.

7. Limitations and Concerns

There are a variety of limitations and concerns with the modification presented. Most notably is the fact that this style of modification does little to nothing to secure the system against exploits, but instead is simply an attempt to invalidate an assumption that might be made by an exploit developer thus preventing the exploit (or at least the first round of POC exploits) from working properly. This means the modification will not offer the same levels of protection that widely used techniques such as ASLR, DEP, stack canaries, and others are able to provide users at least in its current form.

Similarly, only a selection of bugs and exploitation techniques are directly affected as a result of this modification, and even a smaller selection of those that are based upon heap memory. Double-frees, use after free, heap sprays, ROP and JOP techniques, and many other aspects of an exploit remain largely unaffected. It is possible that a motivated attacker with knowledge that a target was utilizing such modifications could simply rebuild *libc* accordingly, and create a new exploit with appropriate offsets given this new reference *libc*. Even the static addition of 16 bytes to each request could be better suited by the addition of a random number of bytes, and this logic may need be placed in other functions (e.g., *realloc*, *mmap*, etc).

Aside from preventing exploits from working successfully, this modification runs the risk of preventing legitimate applications from running properly. While the authors have yet to identify a case where a legitimate application would break from this modification, it is possible that systems with specific memory requirements and constraints might have implicit assumptions about how *libc* handles their requests for memory. Perhaps even a better way to artificially modify the heap would be through adding unused field to the *malloc_chunk* structure to make it take more space. Regardless of the specific method used, any such modification should undergo thorough analysis and testing prior to being suggested for real-world application.

It is worth noting that addressing many of the above limitations and concerns is outside the scope of this paper due to the exploratory nature of the paper. Instead, the goal here is to present a simple modification that might prove to be interesting for future research, and offer valuable insight into how simple modifications could be made to address some inherent weaknesses in homogeneous systems (e.g., by providing each system with its own unique copy of *libc*). Even if the end result is that this style of modification does not offer the same level of protection as ASLR or DEP have done by preventing the successful exploitation of a certain class of vulnerabilities, it is still useful at increasing the effort required by an attacker to rebuild an exploit.

8. Conclusions

It is worth noting that the authors do not believe that such a simple modification to *glibc* will offer significant security to an attacker that has knowledge of this modification. A practical application of this concept that offers any real amount of security would almost certainly require adding a randomized value to the bytes requested in each *malloc* call (although this could quickly drain system entropy), and require more analysis of unintended effects across a range of systems. The current technique described may not offer any increase in security when an attacker knows of its use. As such, the techniques for dynamically altering heap layout presented in this paper are not intended to be implementation-ready.

The goal of this effort was to demonstrate that the heap layout for an arbitrary binary could be easily modified with little effort, and that this may prove useful at reducing the effectiveness of some exploits. Since exploit developers often use hardcoded offsets, lookups, profiling techniques, etc., when creating an exploit for a vulnerability – which they can do given a known target – it is possible that by invalidating the assumptions about the underlying system we can prevent the first round of exploits from affecting a given system. We expect significant additional effort would be necessary to determine if this technique produces a system that is resilient to some class of vulnerabilities even when attackers are aware of such modifications, such as how ASLR and DEP are able to prevent successful exploitation of some classes of vulnerabilities. However, even without this strong guarantee, such a technique may still offer good-enough levels of security to the average user, as this style of modification would require an attacker expend extra time and effort to modify an exploit.

9. References

[1] AWS Marketplace. Retrieved from https://aws.amazon.com/marketplace/ on June 3, 2015.

[2] Bevan. Drupal Groups. Retrieved from https://groups.drupal.org/node/448073 on June 3, 2015.

[3] Scut and Team Teso. (2001) Exploiting format string vulnerabilities. Retrieved from https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf on June 1, 2015.

[4] Aleph One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, 1996.

[5] Crispan Cowan et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Usenix Security*, vol. 98, pp. 63-78, 1998.

[6] Doug Lea and Wolfram Gloger. (1996) A memory allocator. Retrieved from http://g.oswego.edu/dl/html/malloc.html on Jne 3, 2015.

[7] Michel Kaempf, "Vudo malloc tricks," *Phrack Magazine*, vol. 11, no. 57, 2001.

[8] "Once upon a free()," *Phrack Magazine*, vol.

11, no. 57, 2001.

[9] "Advanced Doug Lea's malloc exploits," *Phrack Magazine*, vol. 11, no. 62, 2003.

[10] Phantsmal Phantasmagoria. (2005) Bugtraq mailinglist. Retrieved from http://www.securityfocus.com/archive/1/413007/30/0/threaded

[11] "Malloc Des-Maleficarum," *Phrack Magazine*, vol. 13, no. 66, 2005.

[12] Chris Evans. (2014) Project Zero. Retrieved from http://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html on May 15, 2015.

[13] Hovav Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86).," *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 552-561, 2007.

[14] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang, "Jump-oriented programming: a new class of code-reuse attack.," *In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 30-40, 2010.

[15] Minh Tran et al., "On the expressiveness of return-into-libc attacks.," *Recent Advances in Intrusion Detection*, pp. 121-141, 2011.

[16] Worawit Wang. (2014) Exploit for Samba vulnerabilty (CVE-2015-0240). Retrieved from https://gist.github.com/worawit/051e881fc94fe4a49295 on May 12, 2015.

[17] Chris Evans. (2014) Google Security Research. Retrieved from https://code.google.com/p/google-security-research/issues/detail?id=96 on May 12, 2015.