Run-time Detection of Heap-based Overflows

William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur
– University of California, Santa Barbara

ABSTRACT

Buffer overflows belong to the most common class of attacks on today's Internet. Although stack-based variants are still by far more frequent and well-understood, heap-based overflows have recently gained more attention. Several real-world exploits have been published that corrupt heap management information and allow arbitrary code execution with the privileges of the victim process.

This paper presents a technique that protects the heap management information and allows for run-time detection of heap-based overflows. We discuss the structure of these attacks and our proposed detection scheme that has been implemented as a patch to the GNU Lib C. We report the results of our experiments, which demonstrate the detection effectiveness and performance impact of our approach. In addition, we discuss different mechanisms to deploy the memory protection.

Introduction

Buffer overflow exploits belong to the most feared class of attacks on today's Internet. Since buffer overflow techniques have reached a broader audience, in part due to the Morris Internet worm [1] and the Phrack article by AlephOne [2], new vulnerabilities are being discovered and exploited on a regular basis. A recent survey [3] confirms that about 50% of vulnerabilities reported to CERT are buffer overflow related.

The most common type of buffer overflow attack is based on stack corruption. This variant exploits the fact that the return addresses for procedure calls are stored together with local variables on the program's stack. Overflowing a local variable can thus overwrite a return address, redirecting program flow when the function returns. This potentially allows a malicious user to execute arbitrary code.

Recently, however, buffer overflows that corrupt the heap have gained more attention. Several CERT advisories [4,5] describe exploits that affect widely deployed programs. Heap-based overflows can be divided into two classes: One class [6] comprises attacks where the overflow of a buffer allocated on the heap directly alters the content of an adjacent memory block. The other class [7,8] comprises exploits that alter management information used by the memory manager (i.e., malloc and free functions). Most malloc implementations share the behavior of storing management information within the heap space itself. The central idea of the attack is to modify the management information in a way that will allow subsequent arbitrary memory overwrites. In this way, return addresses, linkage tables or application level data can be altered. Such an attack was first demonstrated by Solar Designer [9].

This paper introduces a technique that protects the management information of boundary-tag-based heap managers against malicious or accidental modification. The idea has been implemented in Doug Lea's malloc for GNU Lib C, version 2.3 [10], utilized by Linux and Hurd. It could, however, be easily extended to other systems such as various free BSD distributions. Using our modified C library, programs are protected against attacks that attempt to tamper with heap management information. It also helps to detect programming errors that accidentally overwrite memory chunks, although not as complete and verbose as available memory debuggers. Program recompilation is not required to enable this protection. Every application that is dynamically linked against Lib C is secured once our patch has been applied.

Related Work

Much research has been done on the prevention and detection of stack-based overflows. A well-known result is StackGuard [11], a compiler extension that inserts a 'canary' word before each function return address on the stack. When executing a stack-based attack, the intruder attempts to overflow a local buffer allocated on the stack to alter the return address of the function that is currently executing. This might permit the attacker to redirect the flow of execution and take control of the running process. By inserting a canary word between the return address and the local variables, overflows that extend into the return address will also change this canary and thus, can be detected.

There are different mechanisms to prevent an attacker from simply including the canary word in his overflow and rendering the protection ineffective. One solution is to choose a random canary value on process startup (i.e., on exec) that is infeasible to guess. Another solution uses a terminator canary that consists of four different bytes commonly utilized as string terminator characters in string manipulation library functions (such as strcpy). The idea is that the attacker is required to insert these characters in the string used to overflow the buffer to overwrite the canary and remain undetected. However, the string manipulation functions will stop when encountering a terminator character and thus, the return address remains intact.

A similar idea is realized by StackShield [12]. Instead of inserting the canary into the stack, however, a second stack is kept that only stores copies of the return addresses. Before a procedure returns, the copy is compared to the original and any deviations lead to the abortion of the process. Stack-based overflows exploit the fact that management information (the function return address) and data (automatic variables and buffers) are stored together. StackGuard and StackShield are both approaches to enforcing the integrity of in-band management information on the stack. Our technique builds upon this idea and extends the protection to management information in the heap.

Other solutions to prevent stack-based overflows are not enforced by the compiler but implemented as libraries. Libsafe and Libverify [13,14] implement and override unsafe functions of the C library (such as strcpy, fscanf, getwd). The safe versions estimate a safe boundary for buffers on the stack at run-time and check this boundary before any write to a buffer is permitted. This prevents user input from overwriting the function return address.

Another possibility is to make the stack segment non-executable [15]. Although this does not protect against the actual overflow and the modification of the return address, the solution is based on the observation that many exploits execute their malicious payload directly on the stack. This approach has the problem of potentially breaking legitimate uses such as functional programming languages that generate code during run-time and execute it on the stack. Also, gcc uses executable stacks as function trampolines for nested functions and Linux uses executable user stacks for signal handling. The solution to this problem is to detect legitimate uses and dynamically re-enable execution. However, this opens a window of vulnerability and is hard to do in a general way.

Less work has been done on protecting heap memory. Non-executable heap extensions [16, 17] that operate similar to their non-executable stack cousins have been proposed. However, they do not prevent buffer overflows from occurring and an attacker can still modify heap management information or overwrite function pointers. They also suffer from breaking applications that dynamically generate and execute code in the heap.

Systems that provide memory protection are memory debuggers, such as Valgrind [18] or Electric Fence [19]. These tools supervise memory access (read and write) and intercept memory management calls (e.g., malloc) to detect errors. These tools use an approach similar to ours in that they attempt to maintain the integrity of the utilized memory. However, a check is inserted on every memory access, while our approach only performs a check when allocating or deallocating memory chunks. Memory debuggers effectively prevent unauthorized memory access and stop heap-based buffer overflows. Yet, they also impose a serious performance penalty on the monitored programs, which often run an order of magnitude slower. This is not acceptable for most production systems.

A recent posting on bugtraq pointed to an article [20] that discusses several techniques to protect stack and heap memory against overflows. The presented heap protection mechanism follows similar ideas as our work as it aims at protecting heap management information. However, no details were provided and no implementation or evaluation of their technique exists.

A possibility of preventing stack-based and heap-based overflows altogether is the use of type-safe languages such as Java. Alternatively, solutions have been proposed [21] that provide safe pointers for C. All these systems can only be attacked by exploiting vulnerabilities [22, 23] in the mechanisms that enforce the type safety (e.g., bytecode verifier). Note, however, that safe C systems typically require new compilers and recompilation of all applications to be protected.

Technique

Heap Management in GNU Lib C (glibc)

The C programming language provides no built-in facilities for performing common operations such as dynamic memory management, string manipulation or input/output. Instead, these facilities are defined in a standard library, which is compiled and linked with user applications. The GNU C library [10] is such a library that defines all library functions specified by the ISO C standard [24], as well as additional features specific to POSIX [25] and extensions specific to the GNU system [26].

Two kinds of memory allocation, static and automatic, are directly supported by the C programming language. Static allocation is used when a variable is declared as static or global. Each static or global variable defines one block of space of a fixed size. The space is allocated once, on program startup as part of the exec operation and is never freed. Automatic allocation is used for automatic variables such as a function arguments or local variables. The space for an automatic variable is automatically allocated on the stack when the compound statement containing the declaration is entered, and is freed when that compound statement is exited.

A third important kind of memory allocation, dynamic allocation, is not supported by C variables but is available via glibc functions. Dynamic memory allocation is a technique in which programs determine during run-time where information should be stored. It is needed when the amount of required memory or when the lifecycle of memory usage depends on factors that are not known a-priori. The two basic functions provided are one to dynamically allocate a block of memory (malloc), and one to return a previously allocated block to the system (free). Other routines (such as calloc, realloc) are then implemented on top of these two procedures.

GNU Lib C uses Doug Lea's memory allocator dlmalloc [27] to implement the dynamic memory allocation functions. dlmalloc utilizes two core features, boundary tags and binning, to manage memory requests and releases on behalf of user programs.

Memory management is based on 'chunks,' memory blocks that consist of application usable regions and additional in-band management information. The in-band information, also called boundary tag, is stored at the beginning of each chunk and holds the sizes of the current and the previous chunk. This allows for coalescing two bordering unused chunks into one larger chunk, minimizing the number of unusable small chunks as a result of fragmentation. Also, all chunks can be traversed starting from any known chunk in either a forward or backward direction.

Chunks that are currently not in use by the application (i.e., free chunks) are maintained in bins, grouped by size. Bins for sizes less than 512 bytes each hold chunks of only exactly one size; for sizes equal to or greater than 512 bytes, the size ranges are approximately logarithmically increasing. Searches for available chunks are processed in smallest-first, best-fit order, starting at the appropriate bin depending on the memory size requested. For unallocated chunks, the management information (boundary tag) includes two pointers for storing the chunk in a double linked list (called free list) associated with each bin. These list pointers are called forward (fd) and back (bk).

On 32-bit architectures, the management information always contains two 4-byte size-information fields (the chunk size and the previous chunk size). When the chunk is unallocated, it also contains two 4-byte pointers that are utilized to manipulate the double linked list of free chunks for the binning.

This basic algorithm is known to be very efficient. Although it is based upon a search mechanism to find best fits, the use of indexing techniques (i.e., binning) and the exploitation of special cases lead to average cases requiring only a few dozen instructions, depending on the machine and the allocation pattern. A number of heuristic improvements have also been incorporated into the memory management algorithm in addition to the main techniques. These include locality preservation, wilderness preservation, memory mapping, and caching [28].

Anatomy of a Heap Overflow Exploit

The use of in-band forward and back pointers to link available chunks in bins exposes glibc's memory management routines to a security vulnerability. If a malicious user is able to overflow a dynamically allocated block of memory, that user could overwrite the next contiguous chunk header in memory. When the overflown chunk is unallocated, and thus in a bin's double linked list, the attacker can control the values of that chunk's forward and back pointers. Given this information, consider the unlink macro used by glibc shown below:

```
    #define unlink(P, BK, FD) {   \
[1]     FD = P->fd;               \
[2]     BK = P->bk;               \
[3]     FD->bk = BK;              \
[4]     BK->fd = FD;              \
    }
```

Intended to remove a chunk from a bin's free list, the unlink routine can be subverted by a malicious user to write an arbitrary value to any address in memory.

In the unlink macro shown above, the first parameter P points to the chunk that is about to be removed from the double linked list. The attacker has to store the address of a pointer (minus 12 bytes, as explained below) in P→fd and the desired value in P→bk. At line [1] and [2], the values of the forward (P→fd) and back pointer (P→bk) are read and stored in the temporary variables FD and BK, respectively. At line [3], FD gets dereferenced and the address located at FD plus 12 bytes (the offset of the bk field within a boundary tag) is overwritten with the value stored in BK. This technique can be utilized, for example, to change an entry in the program's GOT (Global Offset Table) and redirect a function pointer to code of the attacker's choice.

A similar situation occurs with the frontlink macro (shown in Figure 1).

The task of this macro is to store the chunk of size S, pointed to by P, at the appropriate position in the double linked list of the bin with index IDX. FD is initialized with a pointer to the start of the list of the appropriate bin at line [1]. The loop at line [2] searches the double linked list to find the first chunk that is larger than P or the end of the list by following consecutive forward pointers (at line [3]). Note that every list stores chunks ordered by increasing sizes to facilitate a fast smallest-first search in case of memory allocations. When an attacker manages to overwrite the forward pointer of one of the traversed chunks with the address of a carefully crafted fake chunk, he could trick frontlink into leaving the loop (at line [2]) with FD pointing to this fake chunk. Next, the back pointer BK of that fake chunk would be read at line [4] and the integer located at BK plus 8 bytes (8 is the offset of the fd field within a boundary tag) would be overwritten with the address of the chunk P at line [5].

The attacker could store the address of a function pointer (minus 8 bytes) in the bk field of the fake chunk, and therefore trick frontlink into overwriting

```
    #define frontlink(A, P, S, IDX, BK, FD) {      \
        ...
[1] FD = start_of_bin(IDX);
[2]     while ( FD != BK && S < chunksize(FD) ) {  \
[3]         FD = FD->fd;                           \
        }                                          \
[4]     BK = FD->bk;                               \
        ...
[5]     FD->bk = BK->fd = P;                       \
```

Figure 1: frontlink Macro.

this function pointer with the address of the chunk P at line [5]. Although this macro does not allow arbitrary values to be written, the attacker may be able to store valid machine code at the address of P. This code would then be executed the next time the function pointed to by the overwritten integer is called.

```
struct malloc_chunk
{
    INTERNAL_SIZE_T prev_size;
    INTERNAL_SIZE_T size;
    struct malloc_chunk *bk;
    struct malloc_chunk *fd;
};
```

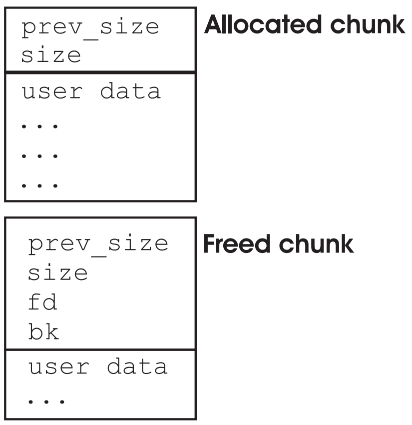

Figure 2: Original memory chunk structure and memory layout.

A variation on the heap overflow exploit described above is also possible, involving the manipulation of a chunk's size field instead of its list pointers. An attacker can supply arbitrary values to an adjacent chunk's size field, similar to the manipulation the list pointers. When the size field is accessed, for example during the coalescing of two unused chunks, the heap management routines can be tricked into considering an arbitrary location in memory, possibly under the attacker's control, as the next chunk. An attacker can set up a fake chunk header at this location in order to perform an attack as discussed above. If an attacker is, for some reason, unable to write to the list pointers of an adjacent chunk header but is able to reach the adjacent chunk's size field, this attack represents a viable alternative.

Heap Integrity Detection

In order to protect the heap, our system makes several modifications to glibc's heap manager, both in the structure of individual chunks as well as the management routines themselves.

```
struct malloc_chunk
{
  INTERNAL_SIZE_T magic;
  INTERNAL_SIZE_T __pad0;
  INTERNAL_SIZE_T prev_size;
  INTERNAL_SIZE_T size;
  struct malloc_chunk *bk;
  struct malloc_chunk *fd;
};
```

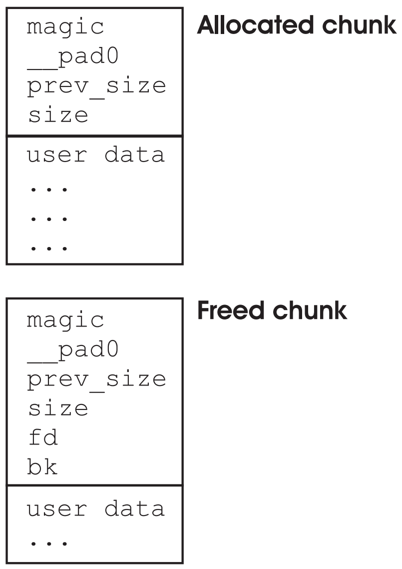

Figure 3: Modified memory chunk structure and memory layout.

Figure 2 depicts the original structure of a memory chunk in glibc.

The first element in protecting each chunk's management information is to prepend a canary to the chunk structure, as shown in Figure 3. An additional padding field, __pad0, is also added (dlmalloc requires the size of a header of a used chunk to be a power of two). The canary contains a checksum of the chunk header seeded with a random value, described below.

The second necessary element of our heap protection system is to introduce a global checksum seed value, which is held in a static variable (called __heap_magic). This variable is initialized during process startup with a random value, which is then protected against further writes by a call to mprotect. This is in contrast to stack protection schemes [29] that rely on repetitive calls to mprotect; since we only require a single invocation during process startup, we do not suffer from any related run-time performance loss associated with other schemes.

The final element of the heap protection system is to augment the heap management routines with code to manage and check each chunk's canary. Newly allocated chunks to be returned from malloc have their canary initialized to a checksum covering their memory location and size fields, seeded with the global value of __heap_magic. Note that the checksum function does not cover the list pointer fields for allocated chunks, since these fields are part of the chunk's user data section. The new chunk is then released to the application.

When a chunk is returned to the heap management through a call to free, the chunk's canary is checked against the checksum calculation performed when the chunk was released to the application. If the stored value does not match the current calculation, a corruption of the management information is assumed. At this point, an alert is raised, and the process is aborted. Otherwise, normal processing continues; the chunk is inserted into a bin and coalesced with bordering free chunks as necessary. Any free list manipulations which take place during this process are prefaced with a check of the involved chunks' canary values. After the deallocated chunk has been inserted into the free list, its canary is updated with a checksum covering its memory location, size fields, and list pointers, again seeded with the value of __heap_magic.

The elements described above effectively prevent writes to arbitrary locations in memory by modifying a chunk's header fields without being detected, whether through an overflow into or through direct manipulation of the chunk header fields. Each allocated chunk is protected by a randomly-seeded checksum over its memory location and size fields, and each free chunk is protected by a randomly-seeded checksum over its memory location, size fields, and list pointers. Each access of a list pointer is protected by a check to insure that the integrity of the pointers has not been violated. Also, each use of the size field is protected. Furthermore, the checksum seed has been protected against malicious writes to guarantee that it cannot be overwritten with a value chosen by the attacker.

As a beneficial side-effect, common programming errors such as unintended heap overflows or double invocations of free are detected by this system as well. A double call to free refers to the situation where a programmer mistakenly attempts to deallocate the same chunk twice. This error is detected due to a checksum mismatch. When the chunk is deallocated for the first time, its canary is updated to a new value reflecting its position on the free list. When the second call to free is executed, the checksum is checked again, with the assumption that it is an allocated chunk. However, since the canary has been updated and the check fails, an alarm is raised.

A limitation of our approach is the fact that we do not address general pointer corruption attacks, such as subversion of an application's function pointers. The system does not guarantee the integrity of user data contained within chunks in the heap; rather, the system guarantees only that the chunk headers themselves are valid.

It is also worth noting that the heap implementation included with glibc already contains functionality that attempts to ensure the integrity of the heap management information for debugging purposes. However, use of the debugging routines incurs significant cost in a production environment. The routines perform a full scan of the heap's free lists and global state during each execution of a heap management function, and include checks unrelated to heap pointer exploitation. Furthermore, there is no guarantee that all attacks are detected. Not all list manipulations are checked, and malicious values could pass integrity checks which are not specifically intended to protect against malicious overflows. Thus, we conclude that the included debugging functionality is not suitable for protecting against the vulnerabilities that we address.

The system described above has been implemented for glibc 2.3 and glibc 2.2.9x, pre-release versions of glibc 2.3 utilized by RedHat 8.0. However, the techniques developed for glibc are easily adaptable to other heap designs, including those shipped with the various BSD derivatives or commercial Unix implementations. Thus, further work is planned to apply this technique to other popular open systems besides glibc.

Evaluation

The purpose of this section is to experimentally verify the effectiveness of our heap protection technique. We also discuss the performance impact of our proposed extension and its stability.

To assess the ability of our protection scheme, we obtained several real-world exploits that perform heap overflow attacks against vulnerable programs. These were

- WU-Ftpd File Globbing Heap Corruption Vulnerability [30] against wuftpd 2.6.0,
- Sudo Password Prompt Heap Overflow Vulnerability [31] against sudo 1.6.3, and
- CVS Directory Request Double Free Heap Corruption Vulnerability [32] against cvs 1.11.4.

In addition, we used two proof-of-concept programs presented in [8] that demonstrate examples of the exploit techniques using the unlink and the frontlink macro, respectively. We also developed a variant of the unlink exploit to demonstrate that dlmalloc's debugging routines can be easily evaded and do not provide protection comparable to our technique.

All vulnerable programs were run under RedHat Linux 8.0. The exploits have been executed three times, once with the default C library (i.e., glibc 2.2.93), once with the patched library including our heap integrity code, and once with the default C library and enabled debugging. The third run was performed to determine the effectiveness of the built-in debugging mechanisms in detecting heap-based overflows.

Table 1 shows the results of our experiments. A column entry of 'shell' indicates that an exploit was successful and provided an interactive shell with the credentials of the vulnerable process. A 'segfault' entry indicates that the exploit successfully corrupted the heap, but failed to run arbitrary code (note that it might still be possible to change the exploit to gain elevated privileges). 'aborted' means that the memory corruption has been successfully detected and the process has been terminated.

The results show that our technique was successful in detecting all corruptions of in-bound management information, and safely terminated the processes. Note that the built-in debugging support is also relatively effective in detecting inconsistencies, however, it does not offer complete protection and imposes a significantly higher performance penalty than our patch.

The performance impact of our scheme has been measured using several micro- and macro-benchmarks. We are aware of the fact that the memory management routines are an important part of almost all applications, and therefore, it is necessary to implement them efficiently. It is obvious that our protection approach inflicts a certain amount of overhead, but we also claim that this overhead is tolerable for most real-world applications and is easily compensated for by the increase in security.

To get a baseline for the worst slowdown that can be expected, we wrote a simple micro-benchmark that allocates and frees around four million (to be more precise, 2^{22}) objects of random sizes between 0 and 1024 bytes in a tight loop. The maximum size of 1024 was chosen to obtain a balanced distribution of objects in dedicated bins (for chunks with sizes less than 512 bytes) and objects in bins that cover a range of different sizes (for chunks with sizes greater than or equal to 512 bytes). We also utilized the dynamic memory benchmark present in the AIM 9 test suite [33]. Table 2 shows the average run-time in milliseconds over 100 iterations for the two micro-benchmarks. We provide results for a system with the default glibc, the glibc with heap protection and the glibc with debugging.

For more realistic measurements that reflect the impact on real-world applications, we utilized Mindcraft's WebStone [34] and OSDB [35]. WebStone is a client-server benchmark for HTTP servers that issues a number of HTTP GET requests for specific pages on a Web server and measures the throughput and response latency of each HTTP transfer. OSDB (open source database benchmark) is a benchmark that evaluates the I/O throughput and general processing power of GNU Linux systems. It is a test suite built on AS3AP, the ANSI SQL Scalable and Portable Benchmark, for evaluating the performance of database systems.

Figure 4 and Figure 5 show the throughput and the response latency measurements for an increasing number of HTTP clients in the WebStone benchmark, for both the default glibc and the patched version. We used an Intel Pentium 4 with 1.8 GHz, 1 GB RAM, Linux RedHat 8.0, and a 3COM 905C-TX NIC for the experiments, running Apache 2.0.40. It can be seen that even for hundred simultaneous clients, virtually no performance impact was recorded. Similar results have been obtained for OSDB 0.15.1. The following Table 3 shows the measurements for 10 parallel clients that used our test machine (the same as above) to full capacity, running a PostgreSQL 7.2.3 database. The results show the total run-time in seconds for the single-user and multi-user tests.

We also attempted to assess the stability of the patched library over an extended period in time. For this

Package	glibc	glibc + heap prot.	glibc + debugging
WU-Ftpd	shell	aborted	aborted
Sudo	shell	aborted	aborted
CVS	segfault	aborted	aborted
unlink	shell	aborted	aborted
frontlink	shell	aborted	aborted
debug evade	shell	aborted	shell

Table 1: Detection effectiveness.

Package	glibc	glibc + heap prot.	glibc + debugging
Loop	1,587	2,033 (+ 28%)	2,621 (+ 65%)
AIM 9	5,094	5,338 (+ 5%)	7,603 (+ 49%)

Table 2: Micro-Benchmarks.

purpose, the patch was installed on the Lab's web server (running Apache 2.0.40) and CVS server (running cvs 1.11.60). A patched library was also used on two desktop machines, running RedHat 8.0 and Gentoo 1.4, respectively. Although the web server only receives a small number of requests, the CVS server is regularly used for our software development and the desktop machines are the workstations of two of the authors. All machines were stable and have been running without any problems for a period of several weeks.

Package	glibc	glibc + heap prot.
OSDB	6,015	6,070 (+ 0.91%)

Table 3: OSDB benchmark.

Installation

Several methods of deploying our heap protection system have been developed, in order to accommodate various system environments and levels of desired protection. Many important security mechanisms are not applied because of the complexity and the required effort during setup. We provide different avenues that range from the installation of a pre-compiled package (with minimal effort) to a complete source rebuild of glibc.

One method is to download and install our library modifications as a source patch against glibc. Administrators can select the version appropriate to their system and apply it against a pristine glibc source tree before proceeding with the usual glibc source installation procedure. Source-based distributions, such as Gentoo Linux, can also easily incorporate these patches into their packaging system.

A second method of deploying is to create packages for various distributions of Linux that replace the system glibc image with a version containing our modifications (such as RedHat RPMs). The advantage of this approach is that virtually all applications on the target machine will be automatically protected against

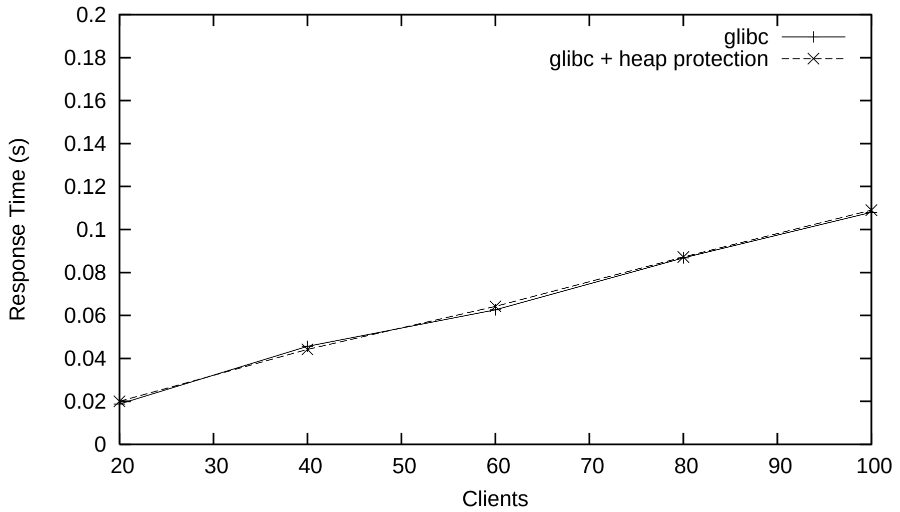

Figure 4: HTTP client response time.

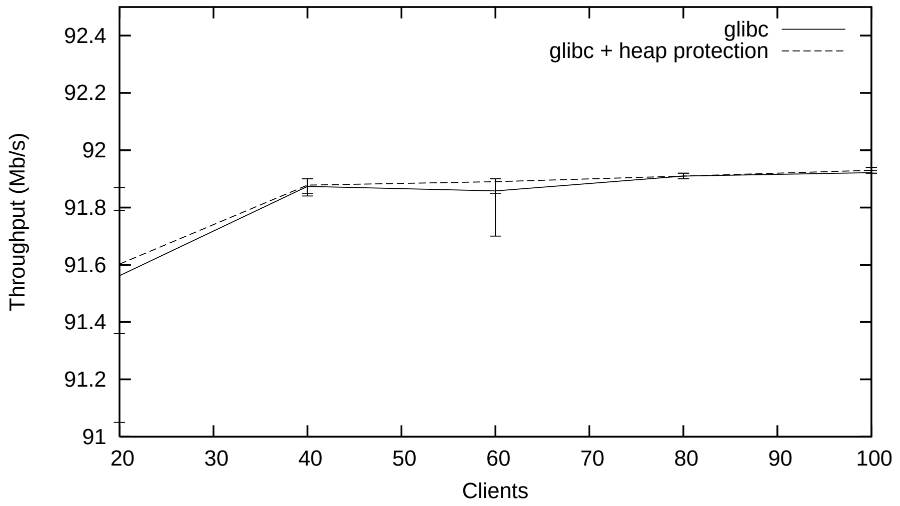

Figure 5: HTTP client throughput.

heap overflow exploitation, with the exception of those applications which are statically linked against glibc or perform their own memory management. A possible disadvantage is that these applications will also experience some level of performance degradation, which could be prohibitive in some high-performance environments.

A third method of deploying our heap protection system uses packages that install a protected glibc image alongside the existing image, instead of replacing the system's glibc image altogether. A script is provided that utilizes the system loader's LD_PRELOAD functionality to substitute the protected glibc image for the system image for an individual application. This allows an administrator to selectively enable protection only for certain applications (e.g., an administrator may not feel it necessary to protect applications which cannot be executed remotely, and therefore may wish to only protect those applications which are network-accessible). This is also a suitable path for admins that are afraid of potentially destabilizing their entire system by performing a system-wide deployment of a heap modification which has not undergone the extensive real-world testing that standalone dlmalloc has.

All of the described installation methods are documented in detail on our website, located at http://www.cs.ucsb.edu/~rsg/heap/ . Packages for various popular distributions and source patches can be downloaded as well.

Conclusions

This paper presents a technique for detecting heap-based overflows that tamper with in-band memory management data structures. We discuss different ways to mount such attacks and show our mechanism to detect and prevent them. We implemented a patch for glibc 2.3 that extends the utilized data structures with a canary that stores a checksum over the sensitive data. This checksum calculation involves a secret seed that makes it infeasible for an intruder to guess or fake the canary in an attack.

Experience shows that system administrators are often reluctant to adopt security measures in the systems they administer. Installing new tools may require significant effort to understand how to best apply the technology in the administrator's network, as well as investment in training end users. Additionally, applying a new tool may interfere with existing critical systems or impose unacceptable run-time overhead. This paper introduces a heap protection mechanism that increases application security in a way that is nearly transparent to the functioning of applications and is invisible to users. Applying the system to existing installations has few drawbacks. Recompilation of applications is rarely required, and the system imposes minimal overhead on application performance.

Acknowledgments

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Army Research Office, or the U.S. Government.

Author Information

Christopher Kruegel is working as a research postgraduate in the Reliable Software Group at the University of California, Santa Barbara. Previously, he was an assistant professor at the Distributed Systems Group at the Technical University Vienna. Kruegel holds the M.S. and Ph.D. degrees in computer science from the Technical University Vienna. His research focus is on network security, with an emphasis on intrusion detection. You can contact him at chris@cs.ucsb.edu .

Darren Mutz is a doctoral student in the Computer Science department at the University of California, Santa Barbara. His research interests are in network security and intrusion detection. From 1997 to 2001 he was employed as a member of technical staff in the Planning and Scheduling Group at the Jet Propulsion Laboratory where he engaged in research efforts focused on applying AI, machine learning, and optimization methodologies to problems in space exploration. He holds a B.S. degree in Computer Science from UCSB and can be contacted at dhm@cs.ucsb.edu .

William Robertson is a first-year PhD student in the Computer Science department at the University of California, Santa Barbara. His research interests include intrusion detection, hardening of computer systems, and routing security. He received his B.S. degree in Computer Science from UC Santa Barbara, and can be reached electronically at wkr@cs.ucsb.edu .

Fredrik Valeur is currently a Ph.D. student at UC Santa Barbara. He holds a Sivilingenioer degree in Computer Science from the Norwegian University of Science and Technology. His research interests include intrusion detection, network security and network scanning techniques. He can be contacted at fredrik@cs.ucsb.edu .

References

[1] Spafford, E., "The Internet Work Program," *Analysis Computer Communication Review*, 1998.

[2] AlephOne, *Smashing the Stack for Fun and Profit*, http://www.phrack.org/phrack/49/P49-14 .

[3] Wilander, J. and M. Kamkar, "Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," *10th Network and Distributed System Security Symposium*, 2003.

[4] *CERT Advisory CA-2002-11*, "Heap Overflow in Cachefs Daemon (cachefsd)," http://www.cert.org/advisories/CA-2002-11.html .

[5] *CERT Advisory CA-2002-33*, "Heap Overflow Vulnerability in Microsoft Data Access Components (MDAC)," http://www.cert.org/advisories/CA-2002-33.html .

[6] Conover, M., *w00w00 on Heap Overflows*, http://www.w00w00.org/files/articles/heaptut.txt .

[7] anonymous, *Once upon a free()*, http://www.phrack.org/phrack/57/p57-0x09 .

[8] Kaempf, M., *Vudo malloc tricks*, http://www.phrack.org/phrack/57/p57-0x08 .

[9] Designer, Solar, *JPEG COM Marker Processing Vulnerability in Netscape Browsers*, http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt .

[10] *The GNU C Library*, http://www.gnu.org/software/libc/libc.html .

[11] Cowan, C., et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *7th USENIX Security Conference*, 1998 .

[12] Vendicator, *Stack Shield Technical Info*, http://www.angelfire.com/sk/stackshield/index.html .

[13] Baratloo, A., N. Singh and T. Tsai, *Libsafe: Protecting critical elements of stacks*, http://www.research.avayalabs.com/project/libsafe/index.html .

[14] Baratloo, A., N. Singh and T. Tsai, "Transparent Run-time Defense Against Stack Smashing Attacks," *USENIX Annual Technical Conference*, 2000.

[15] Designer, Solar, *Non-executable stack patch,* http://www.openwall.com .

[16] *RSX: Run-time addressSpace eXtender*, http://www.starzetz.com/software/rsx/index.html .

[17] *PAX: Non-executable heap-segments*, http://pageexec.virtualave.net/index.html .

[18] *Valgrind, an open-source memory debugger for x86-GNU/Linux*, http://developer.kde.org/˜sewardj/index.html .

[19] *Electric Fence – Memory Debugger*, http://www.gnu.org/directory/devel/debug/ElectricFence.html .

[20] Huang, Y., *Protection Against Exploitation of Stack and Heap Overflows*, http://members.rogers.com/exurity/pdf/AntiOverflows.pdf .

[21] Necula, George C., Scott McPeak, and Westley Weimer, "CCured: Type-safe retrotting of legacy code," *29th ACM Symposium on Principles of Programming Languages*, 2002.

[22] Dean, D., E. Felten and D. Wallach, "Java Security: From HotJava to Netscape and Beyond," *IEEE Symposium on Security and Privacy*, 1996.

[23] *The Last Stage of Delirium (LSD), Java and Java Virtual Machine Vulnerabilities and their Exploitation Techniques*, http://www.lsd-pl.net/java_security.html .

[24] *ISO JTC 1/SC 22/WG 14 – C*, http://std.dkuug.dk/JTC1/SC22/WG14/index.html .

[25] *ISO JTC 1/SC 22/WG 15 – POSIX*, http://std.dkuug.dk/JTC1/SC22/WG15/index.html .

[26] *The GNU C Library Manual*, http://www.gnu.org/manual/glibc-2.2.5/libc.html .

[27] Lea, D., *A Memory Allocator*, http://gee.cs.oswego.edu/dl/html/malloc.html .

[28] Wilson, P., M. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," *International Workshop on Memory Management*, 1995.

[29] Chiueh, T., and F. Hsu, "RAD: A Compile-time Solution to Buffer Overflow Attacks," *21st Conference on Distributed Computing Systems*, 2001.

[30] *WU-Ftpd File Globbing Heap Corruption Vulnerability*, http://www.securityfocus.com/bid/3581 .

[31] *Sudo Password Prompt Heap Overflow Vulnerability*, http://www.securityfocus. com/bid/4593 .

[32] *CVS Directory Request Double Free Heap Corruption Vulnerability*, http://www.securityfocus.com/bid/6650 .

[33] *AIM IX Benchmarks*, http://www.caldera.com/developers/community/contrib/aim.html .

[34] *Mindcraft WebStone – The Benchmark for Web Servers*, http://www.mindcraft.com/benchmarks/webstone/ .

[35] *OSDB – The Open Source Database Benchmark*, http://osdb.sourceforge.net .