Hiding Your Wares: Transparently Retrofitting Memory Confidentiality into Legacy Applications

Jamie Levy Bilal Khan
Department of Mathematics and Computer Science
John Jay College of Criminal Justice
New York, NY
Email: {jlevy,bkhan}@jjay.cuny.edu

***Abstract*— 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. In this paper, we describe 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. We describe the design of the library and evaluate its benefits and performance trade-offs.**

I. Introduction

Many people doing secure programming in UNIX or UNIX-like environments are painfully surprised by the existence of `/dev/mem` and `/dev/kmem`. Together, these device files permit the root user to access arbitrary contents of physical memory and kernel memory, respectively—byte addresses in `/dev/mem` are interpreted as physical memory addresses, while byte addresses in `/dev/kmem` are interpreted as kernel virtual memory addresses. There is nothing one can do to prevent access to these device files, since from a kernel perspective, root is omnipresent and omniscient. Unfortunately, the ability to read and write to arbitrary memory makes it quite feasible for malicious programs to violate the implicit memory confidentiality and integrity assumptions made by other processes.

The problem of memory confidentiality has received considerable recent attention in the context of the study of "data lifetimes" [1]. Data lifetime researchers have noted that an application's sensitive data is often scattered widely through user and kernel memory, and continues to reside there for indefinite periods of time [2] , even *long after the program terminates*. In contrast, our research here considers the problem of application data confidentiality *during the lifetime of the application*; the data lifetime problem is solved as a corollary.

Certain well-established secure software design principles [6] attempt to address application data confidentiality issues. For example, it is common knowledge [7] that sensitive data such as cryptographic keys and passwords should be zeroed in memory immediately after they are known to be no longer needed. 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 [8]–[11] by triggering program bugs. This paper 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.

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. [4], [14]). One illustrative case of this is Joseph Corey's paper [4] which targets the Honeynet project's Sebek intrusion detection system [12]. Since Sebek is intended to function as an IDS, its effectiveness hinges on remaining hidden from the would-be attacker. Corey illustrates that Sebek can be detected by searching for particular "landmark" patterns in memory, and 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.) In this paper we describe 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 overwrite.

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 providing transparent run-time encryption of heap-allocated memory for both new and existing applications. Our 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.

II. Prior Related Work

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

Memory Debuggers. The Efence [13] memory debugger 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 to serve program heap allocation requests 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 our project because we use a similar mechanism to allow us to interpose between the program and memory. In our case, however, we interpose 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 [5]. 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, Point Guard protects pointers, but not memory contents themselves. It is intended principally to thwart sled address guessing in buffer overflow attacks. In contrast, our project is concerned with protecting the actual contents of memory buffers from being observed by any process other than the application that owns them.

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

III. Design

METAL is designed as a shim library replacing the standard C memory management functions (e.g. `malloc`, `valloc`, `calloc`, `free`, `cfree`, etc.) The shim library memory allocation functions use `mmap` to allocate a protected page in memory, marking the page as inaccessible for reading and writing. Information about memory allocations is maintained in the library internals. 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-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 word-by-word 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 we considered were

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

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

IV. 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-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.

V. 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. In practice, we found no reliable way to enable the attacker to narrow down the value of p. If the attacker is unable to narrow down p then the only viable strategy is to scan the entire memory, which on a machine with $2G$ of memory (and $4K$-sized pages) 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, we found that the attacker was unable to find the secret for well over twice this period of time.

TABLE I
Memory Overhead of METAL

Program	Memory Usage with METAL	Memory Usage w/o METAL
vim	1305 pages	743 pages
emacs	4990 pages	2574 pages
xterm	1355 pages	2182 pages
firefox	18204 pages	9626 pages

A. Simple?

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

B. 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.

C. 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.

D. Performance?

METAL provides memory confidentiality but presents overhead costs both in terms of memory footprint and processing time.

Table I shows the comparative memory resident sizes of four common applications, running with and without METAL. The blowup in memory footprint for such applications is on average less than double, approximately 1.8.

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 $5\times$ and $150\times$ 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

TABLE II

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 III

SECURITY BENEFITS OF METAL

`METAL` `TIMER` (c)	Access Rate (r)	Actual Time to crack	Estimated Time to crack
100000.00μs	100000.00μs	8.5 hrs	7 hrs
100000.00μs	10000.00μs	51.8 min	40 min
100000.00μs	1000.00μs	4.9 min	4 min
100000.00μs	100.00μs	17.4 sec	17 sec
100000.00μs	10.00μs	3.2 sec	3 sec
10000.00μs	100000.00μs	$> 1^*$ days	3 days
10000.00μs	10000.00μs	8.6 hrs	7 hrs
10000.00μs	1000.00μs	53.7 min	40 min
10000.00μs	100.00μs	4.0 min	4 min
10000.00μs	10.00μs	18.5 sec	19 sec
1000.00μs	100000.00μs	$> 2^*$ days	28 days
1000.00μs	10000.00μs	$> 1^*$ days	3 days
1000.00μs	1000.00μs	8.8 hrs	7 hrs
1000.00μs	100.00μs	43.1 min	40 min
1000.00μs	10.00μs	4.4 min	4 min
100.00μs	100000.00μs	$> 3^*$ days	279 days
100.00μs	10000.00μs	$> 2^*$ days	28 days
100.00μs	1000.00μs	$> 1^*$ days	3 days
100.00μs	100.00μs	7.9 hrs	7 hrs
100.00μs	10.00μs	35.7 min	40 min
10.00μs	100000.00μs	$> 4^*$ days	2791 days
10.00μs	10000.00μs	$> 3^*$ days	279 days
10.00μs	1000.00μs	$> 2^*$ days	28 days
10.00μs	100.00μs	$> 1^*$ days	3 days
10.00μs	10.00μs	8.4 hrs	7 hrs

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.

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). We see that 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.

VI. CONCLUSION

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.

ACKNOWLEDGMENTS

The authors would like to thank the Stephen E. Smith Center for Cybercrime, the John Jay College Office of Sponsored Research, and the Center for Computational Sciences at the U.S. Naval Research Laboratory for their support of these research efforts.

REFERENCES

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

[2] 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.

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

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

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

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

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

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

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

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

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

[12] The Honeynet Project. Know your Enemy. http://www.honeynet.org/papers/sebek.pdf

[13] Information and Communication Theory Group What is Electric Fence? http://genlab.tudelft.nl/old/html/helpdesk/software/efence/

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