Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface

Stephen Checkoway
Johns Hopkins University
s@cs.jhu.edu

Hovav Shacham
UC San Diego
hovav@cs.ucsd.edu

Abstract

In recent years, researchers have proposed systems for running trusted code on an untrusted operating system. Protection mechanisms deployed by such systems keep a malicious kernel from directly manipulating a trusted application's state. Under such systems, the application and kernel are, conceptually, peers, and the system call API defines an RPC interface between them.

We introduce *Iago attacks*, attacks that a malicious kernel can mount in this model. We show how a carefully chosen sequence of integer return values to Linux system calls can lead a supposedly protected process to act against its interests, and even to undertake arbitrary computation at the malicious kernel's behest.

Iago attacks are evidence that protecting applications from malicious kernels is more difficult than previously realized.

1. Introduction

The prospect of running trusted tasks or processes on an untrusted operating system is a tantalizing one. Legacy operating systems are complicated and possibly untrustworthy systems, and retargeting an application written for a legacy OS to run on another, supposedly secure new OS may be prohibitively expensive. Retargeting is also not an option if we wish to provide trusted facilities (such as keyboard input [18]) to legacy applications.

But how is it possible to protect a task from the operating system running it? Every interaction between a userland process and the outside world is mediated by the kernel. A malicious kernel could lead a trusted process astray by falsifying its inputs. Furthermore, the kernel runs at higher privilege on the processor, and is specifically charged with managing application memory. A malicious kernel could read an application's secrets from memory, or cause an application to misbehave arbitrarily by modifying its program code.

In the last few years, researchers have proposed systems intended to achieve precisely the objective above: to run trusted code on an untrusted operating system. These proposed systems insinuate a supervisory module at high privilege that cooperates with the trusted application to isolate and protect it from the potentially malicious kernel. The supervisory module may derive its privilege from trusted hardware, as in XOMOS [14] and Flicker [15–17], or from running as a hypervisor, as in Overshadow [6, 22].

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03...$15.00

Listing 1. A Linux program that can be completely compromised by an Iago attack.

```
#include <stdlib.h>
int main() {
    void *p = malloc(100);
}
```

In this paper, we give evidence that protecting applications from malicious kernels is more difficult than previously realized. For concreteness, we make particular reference to the design of Overshadow. We stress, however, that it is not our intention to single out Overshadow. Instead, we consider an abstract Overshadow-style system that prevents a malicious kernel from manipulating the protected application's memory and other resources. Under such a system, the application and kernel are, conceptually, peers, and the system call API defines an RPC interface between them. We illustrate this conceptual relationship in Figure 1, on the next page.

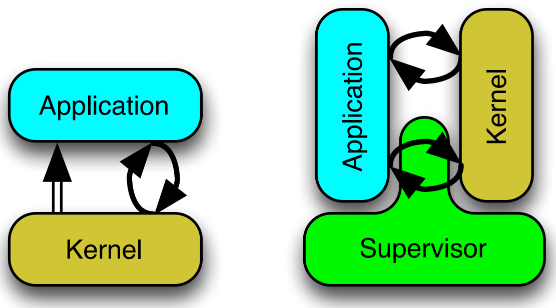
In our main contribution, we describe attacks that a malicious kernel can mount in this model. Specifically, we show how a carefully chosen sequence of integer return values to Linux system calls can lead a supposedly protected process astray. In many cases, including Linux programs as simple as that given in Listing 1, our attacks induce *arbitrary computation* in the protected program. (See Section 4.3 for details of our attack on the program in Listing 1.) We call our attacks *Iago attacks* because our malicious kernel convinces the application to act against its interests simply by communicating with it. From Othello

Some of the systems listed above, such as Flicker, provide only a narrow interface between the trusted component and the untrusted OS, and may therefore not be vulnerable to Iago attacks. The stated design goal of other systems listed above — notably, Overshadow — is protecting legacy applications that make general-purpose system calls and run on untrusted legacy operating systems such as Linux.

Overshadow The Overshadow system, proposed by Chen et al. [6] at ASPLOS 2008, allows legacy applications to run, without modification, on an untrusted kernel.

The fundamental technique introduced by Overshadow is *cloaking*. When the application is running, its memory is mapped normally. At other times, including when the kernel handles a system call on the application's behalf, the application's memory is encrypted and authenticated. Encryption keeps the kernel from reading application memory, and authentication keeps the kernel from modifying application memory. The Overshadow monitor interposes on application-kernel switches to swap between the two views. Overshadow uses virtualization to make cloaking efficient.

A sophisticated system of shims for system calls marshals data between the application and the kernel. Some system calls are



(a) Unprotected **(b)** Protected

Figure 1. Software stack abstraction for (a) unprotected systems and (b) systems protected by an Overshadow-like mechanism. In an unprotected system, the application communicates with the kernel via system calls. Additionally, the kernel is free to read and write application memory at will. In a protected system, the application and kernel are peers which communicate either directly via system calls, as with an unprotected system, or through supervisor intermediation. At no point is the kernel able to directly read or write protected application memory.

modified extensively; for example, Overshadow applications use `mmap()` instead of `read()` and `write()` for secure file I/O. Other system calls, such as `getpid()`, are considered safe and not interposed on [6, Section 6.1].

As we noted above, the Overshadow system's stated security goal is to run general-purpose legacy applications, unmodified, on arbitrary untrusted kernels. Overshadow's authors write:

> Overshadow protects *legacy* applications from the commodity operating systems running them . . . it requires *no changes* to existing operating systems or applications . . . Overshadow is backwards-compatible, protecting a broad range of *unmodified legacy* applications, managed by unmodified commodity operating systems *[emphasis added]*.

Our thesis, in this paper, is that mechanisms for protection against malicious kernels are better suited to protecting specialized processes whose limited interface with the OS has been carefully considered, rather than unmodified, general-purpose, legacy applications.

Subsequent work by Ports and Garfinkel [22] reconsidered and refined the security properties provided by Overshadow. Ports and Garfinkel proposed extensions to Overshadow that prevent a variety of attacks by a malicious OS on a protected application. For example, they observe that incorrect mapping of process IDs can lead to signal misdelivery. To prevent this attack, Ports and Garfinkel associate a "secure process ID" with each process. This secure process ID, which is independent of the usual process ID managed by the OS, is communicated to the parent process on `fork()` and is used for reliable signal delivery.

The attacks considered and protected against are similar to our Iago attacks. However, Ports and Garfinkel are concerned with maintaining semantic guarantees for OS services (e.g., time, entropy, the filesystem, mutual exclusion from critical sections, reliable interprocess communication) in the face of OS misbehavior. By contrast, we show how a malicious kernel can use system call return values in ways not related to the semantic content of these system calls. In some cases, our attacks can cause a protected process to undertake arbitrary computation.

Threat model We consider a trusted application running on a malicious kernel. We assume that the application is unmodified and linked against unmodified system libraries, though the implementation of specific library functions might be modified by the protection system.

Good, clear threat model and plan

The kernel is kept by the protection system from directly reading or manipulating the application's state. The kernel still handles system calls on behalf of the application, however. We assume that it can provide return values of its choice to system calls made by the application. We focus on scalar return values: for example, the `ssize_t` return value of the `read` system call rather than the buffer filled as a result of the read.

The kernel's goal is to subvert the trusted application into disclosing its secrets or behaving otherwise than intended. In the limit, the kernel's goal is to cause the application to undertake arbitrary computation. Simple denial of service is not in scope, because a malicious kernel could always crash or just refuse to boot.

Costs and benefits of abstraction Our threat model abstracts away the details of how applications are protected from a malicious kernel. The benefits of this abstraction are, first, that our findings may be applicable to more than just one protection mechanism and, second, that we are able to run concrete experiments using an off-the-shelf Linux environment. The cost is that we cannot say with certainty that any attacks we identify will actually apply to a specific protection mechanism: It is possible that special-case handling that we have overlooked makes our attacks impossible on some particular system. (We emphatically *do not* claim that we have broken the Overshadow system.)

We believe that the tradeoffs favor studying the problem in the abstract, as we do. In exhibiting attacks that require no other affordance than the system call API, we focus attention on this API as the crux of security in this setting. That is, even a perfect defense mechanism that makes the kernel an untrusted peer to applications is not, by itself, sufficient to secure these applications from attack.

Note that, while we necessarily abstracted Overshadow's protection mechanisms, the actual attacks we describe and mount are absolutely concrete.

Our contributions We make the following contributions:

1. We introduce Iago attacks — attacks in which a malicious kernel induces a protected process to act against its interests by manipulating system call return values — and give a threat model for them.
2. We implement a platform for experimenting with Iago attacks on Linux systems. We add hooks to the Linux kernel and implement a kernel module which contains the bulk of the attack code.
3. We demonstrate Iago attacks against Linux applications. In many cases, our attacks induce arbitrary computation in the protected program. We validate these attacks using our experimentation platform.

The rest of this paper is organized as follows. We begin with a "warmup" and motivating example, showing how an Iago attack that manipulates `getpid()` return values allows connection replay against Apache mod_ssl. We next describe our architecture for experimenting with Iago attacks. Then, we describe our main technical result, an Iago attack that induces arbitrary code execution in any Linux process that uses the `malloc()` C library function. This is followed by a different Iago attack that targets programs using the OpenSSL library. Finally, we consider what makes such Iago attacks possible, and suggest directions for future research.

2. SSL Replay and `getpid()`

An important challenge for trusted applications running on untrusted kernels is communicating with the outside world. For communicating with a local user, such an application will require a trusted path to input and output devices. On the other hand, a trusted application that wishes to communicate with a remote user or service faces exactly the traditional network security problem (with the kernel as an active network adversary). Cryptography is well suited for solving this problem; for example, Chen et al. [6] propose the use of the SSL protocol. Implementing an SSL server on an untrusted kernel is not trivial; indeed, as Ports and Garfinkel observe [22], applications use the kernel as their source of cryptographic randomness. Failure by an application to obtain strong randomness from the kernel can have catastrophic results, as with the Debian PRNG bug [29].

Ports and Garfinkel propose that the trusted supervisor intercept application reads from entropy sources such as `/dev/random` to supply randomness to the application. In this section, we observe that preventing cryptographic randomness vulnerabilities in trusted applications is more subtle than just providing a source of strong randomness. In particular, we show that an Iago attack targeting a seemingly innocuous system call — `getpid()` — allows replay attacks on Apache servers with mod_ssl.

Background: SSL and Apache Before explaining our attack, we briefly recall the SSL protocol and the architecture of mod_ssl.

An SSL protocol interaction begins with a handshake. The handshake allows the client and server to pick session parameters; to establish shared cryptographic secrets; and to verify the identities of one or both against the public-key infrastructure. The shared cryptographic secrets are derived from public nonces contributed by both client and server (called the client random and server random) and from a secret value that, in the most common configuration (RSA key exchange without ephemeral Diffie-Hellman), is contributed by the client alone. (For the details of the SSL handshake, see Rescorla [25].)

As a consequence, the only protection that SSL provides a server against session replay is the server random value. If an SSL server can be made to reuse a server random value from some legitimate connection, an attacker can replay the packets of that connection. The SSL server will accept the connection, verify and decrypt the application-protocol packets, and pass their contents on to higher-layer code for processing. If the higher-layer code does not itself defend against replay, this weakness can allow attackers to repeat actions that authorized users intended to occur just once. For example, a single transfer of money using PayPal could turn into several transfers of the same amount.

SSL functionality in the Apache Web server is implemented by the mod_ssl extension, which itself is built on the OpenSSL library. In the usual configuration, The Apache parent process performs all initialization tasks, then forks child processes that will handle incoming requests. Crucially, the OpenSSL entropy pool used by mod_ssl to generate randomness for the SSL protocol is seeded with entropy from the kernel only in the parent process. Every child process inherits an identical entropy pool when forked. The child processes avoid generating the same randomness by stirring into their entropy pools values that are not secret but that should be distinct: the process ID, obtained with `getpid()`, and the system time in seconds, obtained with `time()`.[1] For more details, see Ristenpart and Yilek [26].

The attack Given the facts above, mounting a connection-replay Iago attack is straightforward. The kernel records the packets sent by a client to an Apache child process. It then fakes a network connection to another child process, and replays the recorded packets to the child. When the child makes `getpid()` and `time()` system calls to stir its entropy pool, the kernel responds with the same values with which it responded to the child that handled the legitimate connection. Ristenpart and Yilek have experimentally verified the feasibility of essentially this attack, in the context of virtual machine rewinding vulnerabilities [26].

If the supervisor provides secure time to trusted applications, the kernel will need to perform replay within a one-second window; otherwise, there is no limit on how long replay is possible.

Different randomness will be generated in subsequent connections to a child process, but the kernel can simply crash each child after a connection, causing the Apache parent to fork a replacement child with the same initial entropy pool.

Lessons While the attack described above allows connection replay against the most popular Unix SSL server, it is more interesting for relying on such seemingly innocuous system calls as `getpid()`. Apache mod_ssl is not using the process ID for its semantic value as an identifier for a process (for example for sending it signals); instead, it is using it as a nonrepeating nonce. A supervisor mechanism for ensuring reliable signal delivery will not necessarily address this non-semantic use of `getpid()`. (Indeed, who is to say that a repeating process ID is unreasonable? The kernel could cause a child to crash and, when the parent forks a new child in its place, give that child the same process ID the crashed child had.)

One might argue that this attack could be prevented by having child processes obtain additional strong entropy from the operating system. But the fact is that Apache as written does not do this, and in this paper we are considering systems to protect off-the-shelf applications. In addition, there are good reasons why Apache is written the way that it is: most importantly, child processes may run with restricted privileges, and may not have access to `/dev/random` or other sources of entropy.

3. Iago infrastructure

In this section, we briefly describe our implementation of a malicious kernel. Readers not interested in these details are encouraged to skip to the next section.

To create a malicious kernel to carry out the Iago attacks, we started with Debian revision 35 of version 2.6.32 of the Linux kernel. In order to ease development of the Iago attacks, we modified the kernel as little as possible, pushing most of the implementation of the attacks to a kernel module that could be easily loaded and unloaded at runtime. The separation allows easy development and testing of the attacks.

Changes in the kernel proper consisted of providing hooks for process creation and termination as well as the addition of a new member, **`struct`** `shadow_state *ss`, in the `mm_struct` structure — the structure which maintains all of the state for a process's memory map. The `shadow_state` structure contains function pointers for the malicious implementation of the `brk`, `mmap2`, and `munmap` system calls. At process creation time, if the kernel module is loaded and wishes to attack the process, it can set `ss` to point to a particular `shadow_state` instance whose function pointer members are initialized to point to the desired, malicious functionality. To enable the use of the standard, nonmalicious functions in the module, the kernel exports symbols corresponding to the "real" functions which can be called as needed.

The implementation of the three system calls is changed to check if the `ss` member is non-`NULL` and if so if the function pointer corresponding to the system call is non-`NULL`. If both are non-`NULL`, then the function pointed to by the pointer is used; otherwise the real function is called. For example, the complete implementation of the `brk` system call is given in Listing 2. The others are similar. Note the calls to `down_write()` and `up_write()`. These are to lock

[1] In cryptographic terms, this is called domain separation.

Listing 2. New implementation of the `brk` system call.

```
SYSCALL_DEFINE1(brk,
                unsigned long, brk)
{
    unsigned long retvalue;
    struct mm_struct *mm;
    struct shadow_state *ss;
    mm = current->mm;
    down_write(&mm->mmap_sem);
    ss = mm->ss;
    if (unlikely(ss != NULL) &&
        ss->brk != NULL)
        retvalue = (*ss->brk)(brk);
    else
        retvalue = real_brk(brk);
    up_write(&mm->mmap_sem);
    return retvalue;
}
```

Table 1. Lines of code for each component of our malicious kernel. The number for the kernel is the sum of the number of lines added (129) and the number of lines deleted (12). The kernel module is separated into the core — which includes the code for the sysfs interface, as well as the process creation and exit hooks — and the profiles described in Sections 4 and 5.

Component	lines of code
kernel	141
module core	354
malloc profile	111
openssl profile	111

and unlock the read/write semaphore that protects the `mm_struct` structure. In fact, a significant fraction of the implementation is concerned solely with avoiding race conditions and deadlocks, including handling the module being unloaded in the middle of an active Iago attack.

The majority of the Iago attacks is implemented as a kernel module. When the module is loaded, it installs hooks for process creation and exit and exports a simple control interface using the sysfs pseudo file-system. The sysfs interface allows executables on disk to be associated with a *profile*.

A profile is the implementation of a particular attack and consists of a malicious implementation of the system calls the attack requires. When a process is created after the module has been loaded, the process creation hook is called. If the executable on disk has been associated with a profile, then the process's `mm->ss` member is set to an appropriately filled `shadow_state` structure. As the program executes, the relevant system calls are handled by the code for the profile as described above.

The effort to construct a malicious kernel from a nonmalicious kernel is relatively minor. Table 1 shows a breakdown of the amount of code written.

In principle, the `read` (or any other system call) could be handled in the same manner. However, since the *behavior* of `read` does not need to change for our attacks, we rely on normal input redirection or socket behavior to supply the necessary data.

Similarly, one could easily modify the kernel to prevent address space layout randomization (ASLR). A process can inhibit the randomization of its children in Linux by calling the `personality()` function with the `ADDR_NO_RANDOMIZE` bit

Table 2. Standard I/O functions which read files [23].

`fgetc()`	`getchar_unlocked()`
`fgets()`	`getdelim()`
`fread()`	`getline()`
`fscanf()`	`gets()`
`getc()`	`scanf()`
`getc_unlocked()`	`vfscanf()`
`getchar()`	`vscanf()`

The paper assumes a spohicated understanding of the Linux kernel that is not adequately described in its introduction.

of the argument set. Since it is easiest to work with a consistent address space layout including stack location, all of our victim programs are launched via a helper program which sets the arguments and environment to a known state, performs input and output redirection, and disables ASLR.

4. Compromising any program using `malloc()`

In this section, we show how any program which uses `malloc()` — including the 4-line program in Listing 1 — can be induced to perform arbitrary code execution by a malicious kernel that behaves exactly like a normal kernel except for some carefully chosen return values for standard Linux system calls. We describe the attack in stages.

4.1 `mmap()` and `read()`

For the first stage, consider the following code fragment

```
p = mmap(NULL, 1024, prot,
         flags, -1, 0);
read(fd, p, 1024);
```

which memory maps a 1024 byte region of memory via the `mmap2` system call and then reads up to 1024 bytes into it from a file descriptor using the `read` system call. This fragment of code is vulnerable to an Iago attack.

Since the kernel is responsible for memory management, a malicious kernel can return an address that is *not* a newly allocated memory region, but rather is an address on the stack. When the `read` occurs, the stack will be overwritten with up to 1024 bytes of the kernel's choice. At this point, a saved return address on the stack may be overwritten and the program can be coerced into executing a return-oriented program [27].

This all seems moot... the kernel can do whatever it wants

4.2 Standard I/O

Most programs do not themselves use the `mmap()` and `read()` functions; however, any program that uses standard I/O functions to read from a file — such as those listed in Table 2 — does. In particular, standard I/O functions like `fread()` perform I/O buffering for performance reasons. A buffer sized to hold one file system block, typically 4096 bytes, is allocated by `mmap()` in the EGLIBC internal function `_IO_file_doallocate()` and filled by the `_IO_new_file_underflow()` function which calls `read()`.

As before, the kernel can respond to the `mmap2` call with the address of a saved return address on the stack and then respond to `read` with a return-oriented program. In this way, any program that performs file input using the standard I/O functions is vulnerable.

4.3 Malloc

By carefully responding to `brk` system calls, a malicious kernel can confuse malloc into writing a single word of the kernel's choice into the application's memory. How this is accomplished depends heavily on the specifics of the operation of the malloc implementation and how it interacts with the system call wrappers in the rest of libc. We describe this in detail below.

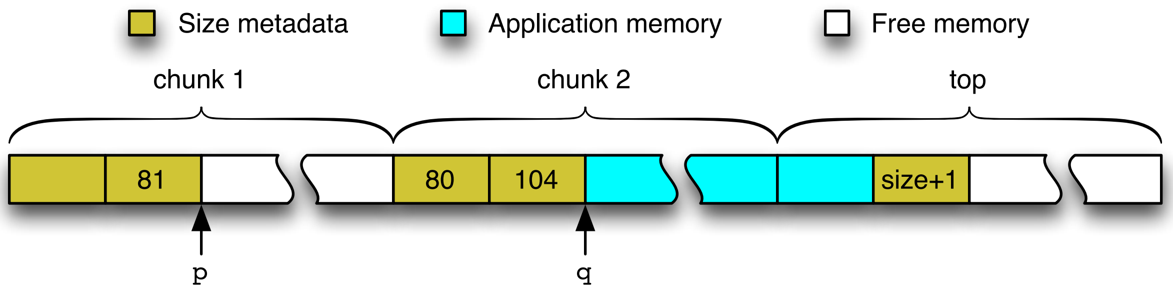


Figure 2. The state of the heap after the following function calls.

```
void *p = malloc(72);
void *q = malloc(100);
free(p);
```

chunk 1 has a size of 80 bytes (72 plus 4 for metadata plus 4 to get an 8 byte alignment) and has the PREV_INUSE bit set since it is the first chunk (hence a size field of 81 rather than 80).

chunk 2 has a size of 104 bytes (100 plus 4 for metadata) and the PREV_INUSE bit is clear, because p was freed, so the size of chunk 1 is stored in prev_size. Since this chunk is in use, the application memory extends into what would otherwise be the prev_size field of the top chunk.

top is the top-most chunk. It is always free and always has the PREV_INUSE bit set.

Note that pointers p and q point 8 bytes after the start of their corresponding chunks.

The version of malloc used in EGLIBC 2.1.2 is a substantially modified version of ptmalloc2 by Wolfram Gloger based on Doug Lea's dlmalloc. EGLIBC's malloc is cleanly separated into upper and lower halves. The upper half is responsible for allocating and freeing regions of memory for the application by requesting a new region of memory from the lower half, splitting and merging free regions, managing a menagerie of free lists, and generally performing the bookkeeping necessary to handle application requests. It implements the public functions specified by the C99 standard [7, Section 7.20.3] including malloc() and free(). This half is, by now, well studied in the literature [1, 3, 4, 13, 21]. The lower half, by contrast, is tasked with claiming and releasing pages of memory from and to the operating system. It is this half that we are most interested in.

Malloc's view of allocated memory is different from the application programmer's. Every region of allocated memory tracked by malloc is called a *chunk*. Broadly speaking, there are three types of chunks, chunks that are in use by the program, free chunks, and the special "top" chunk which can grow and shrink as malloc's lower half requests memory from and returns memory to the system. Each chunk contains the metadata necessary to free the chunk, place it on free lists, and coalesce it with adjacent chunks. (Having inline metadata is not the only way to structure an allocator, see Novark and Berger for a concise overview of several approaches [19, Section 2].) A chunk is defined as

```
struct malloc_chunk {
    size_t prev_size;
    size_t size;
    /* ... */
};
```

where the elided members are for managing doubly-linked lists of chunks. The least-significant bit of the size member is the PREV_INUSE bit. If it is set, then the previous chunk is in use (or is not tracked by malloc). Otherwise, it is free and the prev_size member contains its size. The second-least-significant bit of size is the IS_MMAPPED bit and it is set if the chunk was allocated using mmap(). (The third-least-significant bit is also metadata other than the size but it is not important here.) After a chunk is created to satisfy an application request, malloc() returns the address 8 bytes past the start of the chunk; that is, the address of memory after the size member.[2] This is the view of the allocated memory that the programmer has.

The only member that is always needed when a chunk is in use is the size member. The prev_size member is only needed when the preceding chunk is free. As a result, prev_size can share space with the preceding chunk and the members for managing the linked lists can share space with the application memory. Thus, each chunk has only a 4 byte overhead.

Figure 2 shows the three chunks — chunk 1, chunk 2, and top — that result after several calls to malloc() and free(). First, memory is allocated from the system by the lower half of malloc, described below, to produce top. Then, chunk 1 and chunk 2 are split off from top and pointers to the application region of the chunk is returned to the program. Finally chunk 1 is freed and the PREV_INUSE bit and the prev_size member of chunk 2 are set resulting in the values in memory shown in the figure.

When malloc's upper half needs more memory, because it cannot satisfy a request from the free chunks, for example, it calls the internal function sYSMALLOc(), passing the size of memory it needs to accommodate the malloc() request, including 4 bytes for the size member, and maintaining 8 byte alignment. If sYSMALLOc() can satisfy the request, it will return a pointer to the application memory of a chunk of the requested size as well as potentially modifying the top chunk.

A simplified description of the algorithm used by the internal function sYSMALLOc() is given in Algorithm 1. This omits all error handling not essential for our purposes, allocations on threads other than the main thread, and issues of noncontiguous allocations including applications calling __sbrk() themselves. The set_size() function sets the size member of a chunk; chunk2mem() returns the application's view of the chunk, namely, it returns the address 8 bytes past the beginning of the chunk; and chunk_at_offset(chunk, offset) treats the memory at address chunk + offset as a chunk.

[2] On 64 bit systems, size_t is typically 8 bytes so the address returned by malloc(), in that case, is 16 bytes past the start of the chunk. For concreteness, we focus on the 32 bit case.

Algorithm 1 A simplified version of the `sYSMALLOc` algorithm.

```
1:  function sYSMALLOc($nb$)        ▷ $nb$ is the request size in bytes plus 4 aligned to an 8 byte boundary
2:      if $nb > mmap\_threshold$ then
3:          $size \leftarrow nb + 4$ aligned to a page boundary
4:          $p \leftarrow$ mmap($size$)
5:          if mmap() call succeeded then
6:              set_size($p$, $size$|IS_MMAPPED)
7:              return chunk2mem($p$)
8:      $top\_size \leftarrow$ the size of the top chunk
9:      $size \leftarrow nb + top\_pad + 8 - top\_size$ aligned to a page boundary
10:     $brk \leftarrow$ __sbrk($size$)
11:     if __sbrk() call failed then
12:         Add the size of the current top chunk, $top\_size$, back into $size$ and align to a page boundary
13:         if $size < 1$ MB then
14:             $size \leftarrow 1$ MB
15:         $brk \leftarrow$ mmap($size$)
16:         if mmap() call succeeded then
17:             $top \leftarrow brk$
18:             set_size($top$, $size$|PREV_INUSE)
19:     else
20:         if $brk$ is the end of the top chunk then
21:             set_size($top$, $(size + top\_size)$|PREV_INUSE)        ▷ extend the top chunk by $size$
22:         else                                                          ▷ first call to malloc()
23:             let $correction$ be the number of bytes needed to ensure chunk2mem($brk$) is 8-byte aligned
24:             if $correction > 0$ then
25:                 $brk \leftarrow brk + correction$
26:             $correction \leftarrow correction + top\_size$        ▷ this was subtracted out in line 9
27:             extend $correction$ so that $brk + size + correction$ ends on a page boundary
28:             $snd\_brk \leftarrow$ __sbrk($correction$)
29:             if __sbrk() call failed then        ▷ determine where the end of the allocated memory lies
30:                 $correction \leftarrow 0$
31:                 $snd\_brk \leftarrow$ __sbrk(0)
32:             $top \leftarrow brk$
33:             set_size($top$, $(snd\_brk - brk + correction)$|PREV_INUSE))
34:     $p \leftarrow top$
35:     $size \leftarrow$ the size of the top chunk
36:     if $size > nb + 8$ then
37:         $top \leftarrow$ chunk_at_offset($top$, $nb$)
38:         set_size($p$, $nb$|PREV_INUSE)        ▷ allocate $nb$ bytes
39:         set_size($top$, $(size - nb)$|PREV_INUSE)
40:         return chunk2mem($p$)
41:     return NULL
```

Lines 2–7 handle the case where the requested size nb meets the threshold to be allocated directly by `mmap()`. Lines 8–10 attempt to extend the program's data memory using `__sbrk()` far enough to accommodate the request along with some additional padding. If `__sbrk()` fails, then lines 11–18 resort to allocating at least one megabyte of memory using `mmap()` which will become the new top chunk shortly. In the common case, `__sbrk()` will succeed and will furthermore have extended the space previously allocated by an `__sbrk()`. If so, then the size of the top chunk is set to be the old size plus the size of the newly allocated region; line 21. The first time `sYSMALLOc()` is called, there will have been no previous call to `__sbrk()` and thus no space to extend so lines 23–33 will perform the initial setup which consists of ensuring the beginning and ending alignment of the memory is correct. (This code path is also taken in the event the `__sbrk()` on line 10 failed but the `mmap()` succeeded.) Finally, if any of the allocation paths have succeeded in creating a top chunk that is large enough to satisfy the request, then line 37–40 will split an nb-sized chunk off and return a pointer to the application memory region.

The alignment fixup the first time `sYSMALLOc()` is called in lines 23–33 is to ensure that chunks that are split from the top chunk are 8-byte aligned and that the top chunk ends on a page boundary. We can use the interaction of the three calls to `__sbrk()` (lines 10, 28, and 31) to control where malloc thinks the data memory starts and ends. This is integral to confusing it into writing a word of our choice at a memory location also of our choice. To see how we can accomplish this, we need to look at the details of the `__sbrk()` function, the `__brk()` wrapper function, and the `brk` system call.

At the lowest level, the `brk` system call takes as an argument the requested new program break — the end of the process's data memory — and is supposed to return the break that results from the call. In the special case that the argument is 0, `brk` just returns the current break without changing it. The EGLIBC wrapper function `__brk()` takes the requested break as an argument and returns 0 if the break returned by the system call is at least as great as the requested break and -1 otherwise. EGLIBC maintains a global variable `__curbrk` which is initially `NULL` but is updated with

Algorithm 2 Pseudocode for the `__sbrk()` function.

```
function __sbrk($increment$)
    if __curbrk = NULL then
        __brk(0)
    if $increment = 0$ then
        return __curbrk
    $oldbrk \leftarrow$ __curbrk
    if $oldbrk + increment$ does not overflow then
        if __brk($oldbrk + increment$) = 0 then
            return $oldbrk$
    return $-1$
```

the result of the brk system call in `__brk()`, even if `__brk()` ultimately returns an error.

By contrast, the `__sbrk()` function takes an amount by which the break should be incremented and returns the previous value of the break if it is able to extend the break by at least that amount, otherwise it returns -1. Algorithm 2 contains the pseudocode for `__sbrk()`.

In order to control where malloc thinks the start and end of the data memory region lie, the kernel only needs to respond appropriately to the brk system calls. To see this, assume the kernel wants malloc to think the start of kernel memory is at address S and the end lies at address E and that the first call to `sYSMALLOc()` has argument nb which is less than the threshold for using `mmap()`. Since malloc will ensure that its start of data memory is on an 8-byte boundary, assume that S is also on an 8-byte boundary.

At line 10, `sYSMALLOc()` will call `__sbrk()` passing in some positive increment $size > nb$. Since this is the first time `__sbrk()` has been called, `__curbrk` is `NULL` and so `__brk(0)` is called to set it. At this point, the kernel responds to the brk system call with $S - 1$. Since the increment is positive, `__brk()` is called with argument $S - 1 + size$. The kernel responds to the second brk system call with $S - 1 + size$, exactly as requested and thus `__sbrk()` returns $S - 1$.

Since the `__sbrk()` call succeeded and this is the first call to `sYSMALLOc()`, it will determine that it needs to increase *brk* by 1 on line 25 to reach an 8-byte alignment. It will then call `__sbrk()` a second time (line 28) with an additional correction so that the ending address ends on a page boundary. This causes a third and final brk system call. The kernel returns E. If E is less than the requested break, which it will be for our use, then `__curbreak` will be set to E and `__sbrk()` will return -1. Finally, `__sbrk()` is called a final time (line 31) to determine the end of memory and `__sbrk()` will return E without consulting the kernel. This process is illustrated in Figure 3.

After the region of data memory is determined, `sYSMALLOc()` sets that as the top chunk and then splits off a chunk of size nb to satisfy the request. In particular, line 39 writes $(E - S - nb)|$`PREV_INUSE` to location $S+nb+4$, see Figure 4. By carefully picking the values of S and E, we can cause `sYSMALLOc()` to write a word we choose to any location in memory that has an address congruent to 4 modulo 8.

In particular, the word in memory we wish to overwrite is a saved instruction pointer from a `call` instruction. Fortunately (for the attacker), gcc ensures that the stack pointer is congruent to 0 modulo 16 before every `call` so that the instruction pointer is saved to an address congruent to 12 modulo 16 and thus congruent to 4 modulo 8. The address we choose to write is that of the `_IO_gets()` function — which is the implementation of the `gets()` function — and we write it over the saved instruction

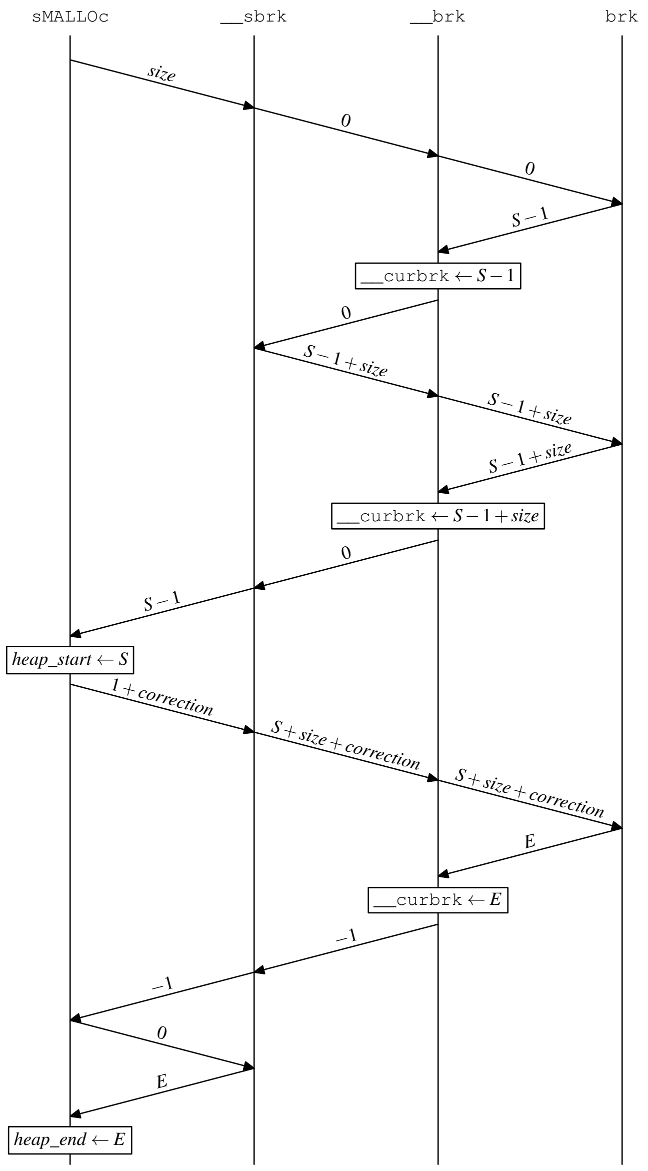


Figure 3. The sequence of function and system calls along with their arguments and the return values used to control which region of memory malloc treats as the heap. Labels on right-facing arrows are arguments; labels on left-facing arrows are return values.

pointer in `_int_malloc()`'s stack frame.[3] In fact, we cannot write the address of `_IO_gets()` because the address is even and ORing `PREV_INUSE` adds one to the address. Fortunately, the first byte in the function is `0x55` which is the opcode for the `push ebp` instruction and can safely be skipped since we will not be returning from this function.

A complete example

As a complete example, consider the program in Listing 1. The request for 100 bytes is increased to 104 bytes for chunk metadata. Since this is already a multiple of 8, $nb = 104$. The `_IO_gets()` function is loaded at address `0xb7ef2010`. The saved instruc-

[3] The code for `sYSMALLOc()` is inlined into `_int_malloc()` which is called by `malloc()`.

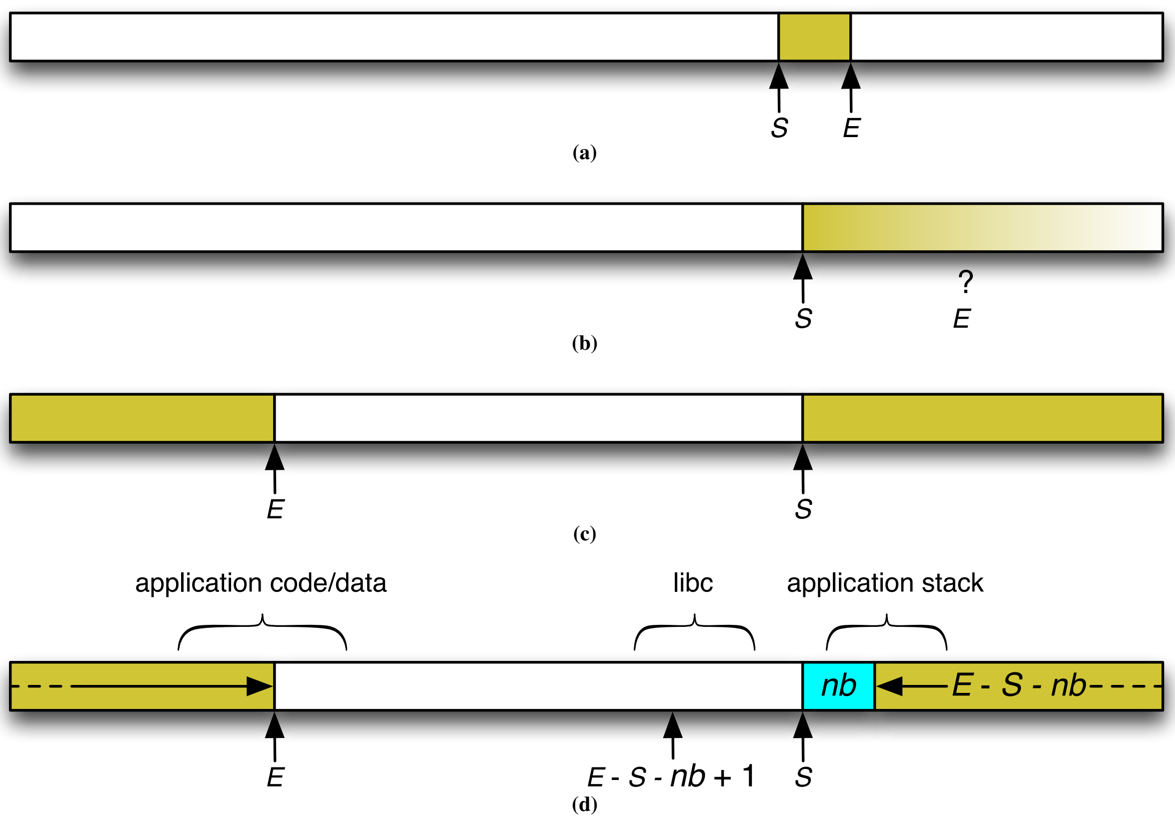


Figure 4. Confusing malloc into overwriting a saved instruction pointer.
During the first call to `sYSMALLOc()`, malloc will request that the break be extended in order to return a chunk of size nb. The first call to `__sbrk()` (line 10) will extend the break and return the old break. At this point, malloc thinks the start of the heap is at location S — the return value from `__sbrk()` and the end of the heap is simply S plus the size it requested the break be extended, as illustrated in (a).
The kernel returns a value that is not 8 byte aligned so malloc increases the start of the heap until it is aligned and requests the break be extended by the corresponding amount using a second call to `__sbrk()` (line 28). The kernel returns a value less than S which causes `__sbrk()` to return a failure. At this point, (b), malloc knows the start of the heap but not the end.
Next, `__sbrk()` is called a third time to determine the end of the heap E, as shown in (c). This happens without calling into the kernel because EGLIBC has recorded the current value of the break from the previous call to `__sbrk()`.
Finally, a chunk of size nb is split off from the heap which causes $E - S - nb + 1$ to be written to address $S + nb + 4$ as shown in (d). By carefully responding to system calls, a saved instruction pointer on the application's stack, at address $S + nb + 4$, can be overwritten with the address of the second byte of a function in libc, namely $E - S - nb + 1$.

tion pointer for `_int_malloc()` is on the stack at location `0xbfffe03c`. Since we want to overwrite the value at that address, we let $S = \texttt{0xbfffe03c} - 104 - 4 = \texttt{0xbfffdfd0}$. And thus $E = S + \texttt{0xb7ef2010} + 104 = \texttt{0x77ef0048}$. After responding to the `brk` system calls as described above, `_int_malloc()` returns to second instruction in the `_IO_gets()` function.

The `_IO_gets()` function calls a series of functions including `_IO_default_uflow()`; `_IO_doallocate()`, which allocates a new buffer via the `mmap2` system call; and finally, `_IO_new_file_underflow()`, which fills the buffer using the `read` system call. The kernel responds to the `mmap2` system call with the address of the saved instruction pointer in `_IO_default_uflow()`'s stack frame, `0xbfffe000`. For, `read`, the kernel fills in the buffer with a return-oriented program.

Table 3 shows the relevant system calls used by the program, their arguments, and how the kernel responds. The arguments and return values for `brk` are addresses; the arguments for `mmap2` and `read` are the sizes; and the return value for `mmap2` is the address. The other arguments are unimportant.

Table 3. Modified system call returns.

System call	Argument	Return value
`brk`	`0`	`bfffdfcf`
`brk`	`c001efcf`	`c001efcf`
`brk`	`c001f000`	`77ef0048`
`mmap2`	`1000`	`bfffe000`
`read`	`1000`	`1e`*

*`read` reads the 30 byte exploit into the buffer.

For this example, our exploit is trivial: It is just a chained return-into-libc attack that chains together a return into the `write()` function which returns into the `_exit()` function. When the program is run with the kernel responding to system calls normally, the program immediately exits. When it is run with the malicious kernel, the kernel causes malloc to overwrite its stack with the exploit payload and the `write()` function outputs a line of text on standard out before exiting via `_exit()`:

```
$ ./victim
Hi there!
```

Arbitrary, Turing-complete computation is possible by changing the exploit to be a more complicated return-oriented program.

5. Compromising OpenSSL

The procedure for compromising malloc given in Section 4 is general purpose and applies to any program that directly or indirectly calls `malloc()`. However, it is only applicable for the first call to `malloc()`. After the initial call, the program break has been established by EGLIBC and the break can only be increased beyond what is requested lest `__sbrk()` fail in `sYSMALLOc()` on line 10. In principle, this is no problem since the kernel can take control and coerce the application to launch an arbitrarily complicated return-oriented program which is able to disclose whatever private information was to remain hidden from the kernel. In practice, emulating enough of the legitimate software to perform the desired malicious action can be quite complicated [5] and taking control further into the program's execution can simplify exploits. In this section, we show how to leverage malloc's fallback to `mmap()` to accomplish this in some cases where the allocated buffer is used as the destination of a `read()` call, similar to the code snippet in Section 4.1.

From Section 4.3, we can control the starting and ending addresses of the program's data region by responding to `brk` system calls. There is an additional restriction on where we can place the end of the data region, which is described below, but the idea is to leverage this ability to control where in a program's execution `sYSMALLOc()` is called a subsequent time. That is, the program makes a number of calls to `malloc()` and `free()` and one of the buffers allocated by `malloc()` is passed to `read()`.

By responding appropriately to `brk`, the kernel arranges for the size of the program's data region to be just large enough that when the program attempts to allocate the region of memory which will be passed to `read()`, malloc is forced to call `sYSMALLOc()`. If the allocation is larger than the *mmap_threshold*, the allocation will be memory mapped (lines 2–7) and so the kernel can return the address of the memory it wishes to overwrite. Otherwise, `__sbrk()` will be called. At this point, the kernel can refuse to increase the break in response to the `brk` system call which will cause `__sbrk()` to fail and `sYSMALLOc()` will fall back on `mmap()` (lines 11–18) and again the kernel can provide the address it wants.

There are several caveats with this method. The first is the restriction on ending addresses for the data region. Due to an assertion early in `sYSMALLOc()`, the end of the data region must be aligned on a page boundary.[4] The second is that the chosen end of the data region must be at an address that is less than the requested one to cause the second call to `__sbrk()` to fail. Thus if we want the end to be at a greater address than requested, we must initially set the end at a smaller address and then handle successive `brk` requests normally until we reach the point we wish it to fail. The final caveat is that a program may allocate a great deal of memory initially and then free it such that subsequent allocations come from the free chunks. The upshot of these caveats is that we cannot always arrange for `brk` to fail for exactly the allocation we wish. However, it may be possible to fail several allocations early.

[4] This assertion appears to be a (mostly harmless) bug in EGLIBC. A comment in the code after the second `__sbrk()` (corresponding to line 28) indicates that the third call is to find the end of memory in the hope that the allocation will still be possible. If the end does not lie on a page boundary, then the next call to `sYSMALLOc()` will (erroneously) abort the program rather than attempt `mmap()` or return `NULL`.

Table 4. Modified system call returns for OpenSSL s_server.

System call	Argument	Return value
`brk`	`0`	`081e4fff`
`brk`	`08205fff`	`08205fff`
`brk`	`08206000`	`081f8000`
`brk`	`0821a000`	`081f8000`
`mmap2`	`1000000`	`bfeff000`

A complete example

While my proposal won't prevent Iago attacks, it will limit the amount of code availabe for ROP progeamming

As an example of the technique of making malloc fall back on `mmap()`, we describe attacking the OpenSSL s_server program. This program (usually started by running the `openssl` binary with the `s_server` option) listens on a specified port for incoming connections and sets up a TLS/SSL connection. Afterward, incoming data is decrypted and written to standard out and data read from standard in is encrypted and sent over the socket. The secret key and certificate used in the TLS/SSL protocol are stored in files on disk.

Under the assumptions of an Overshadow-like system, the kernel would be prevented from reading the contents of the secret key on disk and, of course, it could not read it from `openssl`'s memory during execution. With the help of OpenSSL's s_client program — the companion program to s_server — the kernel will cause s_server to disclose its secret key, in this case, the RSA private exponent.

The first step is to launch OpenSSL s_server.

```
$ openssl s_server -key secret.key \
-cert cert.pem -accept 8080
```

This starts the server listening on port 8080. As before, the kernel responds to the first three `brk` system calls in order to set the length of data memory appropriately as described in Table 4. This causes the top chunk to have `0x13000` bytes of memory, initially.

The next step is to launch OpenSSL s_client with the exploit payload.

```
$ openssl s_client -connect \
localhost:8080 <exploit
```

The client will connect to the server and send the exploit code. After the client connects, the server will allocate `0x4000` bytes of memory for a buffer into which it will read the decrypted data. However, by this point, neither the free chunks nor the top chunk will be large enough to accommodate this allocation, so `sYSMALLOc()` requests more memory via `__sbrk()`. This time, the kernel responds to `brk` by returning the same value as before. This causes `__sbrk()` to return -1 and `sYSMALLOc()` falls back on `mmap()`. The kernel responds to the `mmap2` system call with an address on the stack. The server sets up a TLS connection with the client and then reads the encrypted exploit payload. The payload is decrypted and stored in the buffer which is really part of `SSL_read()`'s stack frame. Rather than returning to the function that called `SSL_read()`, it returns to a simple return-oriented program which calls the `write()` function to write the contents of the private exponent of the secret key to `stderr` and then exits.

6. Discussion and Conclusions

We have introduced Iago attacks: attacks in which a malicious kernel induces a protected process to act against its interests by manipulating system call return values. We have defined a threat model for Iago attacks, implemented a platform for experimenting with Iago attacks, and used this platform to demonstrate Iago attacks against Linux applications, including any application which uses `malloc()`. Some of our attacks induce arbitrary computation in the protected program.

Iago attacks provide a partial answer to an open problem posed by Chen et al.: "The implications of maliciously changing the behavior of seemingly innocuous parts of the system call API, such as those for managing identity and concurrency, are still largely unstudied" [6, Section 2.2].

Iago attacks are evidence that protecting applications from malicious kernels is more difficult than previously realized. We believe that there are several fundamental reasons for this difficulty. First, the system call API was not designed to be an untrusted RPC interface, so unsurprisingly it is a difficult interface to secure. Second, system calls are used at all layers of a program, including the libraries the program links against; securing applications against Iago attacks requires understanding the system calls made at every layer. Third, system calls are frequently used in other ways than for their nominal semantic content; providing a replacement to process IDs for reliable signal delivery does nothing to help OpenSSL's reliance on `getpid()` for entropy stirring.

Ports and Garfinkel [22] suggest that *verifying* that return values are correctly computed is easier than undertaking to compute them, and that a trusted supervisor monitoring the behavior of an untusted kernel can be smaller and simpler than the kernel itself. Our findings do not refute this claim, but they do suggest that the gap between verifying and computing may be smaller than previously realized, at least for the more complex of a kernel's tasks. For some tasks, such as managing virtual memory, verifying return values may require the supervisor to have a complete understanding of a kernel's memory management algorithms and data structures.

Address-space layout randomization makes it hard to exploit memory bugs, but the untrusted kernel is in charge of process creation. How can the supervisor be sure that the kernel isn't placing the process' memory segments in predictable locations?[5] For that matter, what constitutes a reasonable memory layout? At a crucial point, the attack we describe in Section 4.3 overlays an `mmap`ed memory region on the stack, and perhaps this could be noticed and prevented by the supervisor. But there are legitimate reasons that processes would want to map memory on top of an existing mapping. More generally, we believe that variants of our attack are possible without overlapping memory regions. One promising target for such an approach is the stack segment. Oberheide recently demonstrated the possibility of "stack overflow" attacks [20], in which the stack of a program is induced to extend down so far (by means, for example, of a recursive function that parses user input) that it (implicitly) overlaps some other memory segment, leading memory safety guarantees to be violated. Oberheide was able to exhibit stack overflow attacks against real programs run on benign kernels; such attacks would be easier to mount when a malicious kernel decides the layout of the stack and other memory segments in process memory.

Understanding the situations in which verifying return values is easier than computing them, for virtual memory as well as other subsystems represented in the system call API, remains an important open problem. A particularly interesting challenge: Is it always possible to verify a system call return value based on the current state of the system, or are there system call values that can only be provisionally verified and must be checked for consistency with subsequent return values? Put another way, is it or isn't it possible to verify the behavior of a kernel using a constant amount of state as a function of the time a process has been running?

[5] One intriguing possibility is that, on process startup, a shim runs that randomizes the runtime environment. But this is no silver bullet; past work has repeatedly shown that attackers are able to adjust to uncertainty about their target's memory layout; see, e.g., Sotirov and Dowd [28].

One possibility is that running arbitrary applications on an untrusted kernels is too ambitious a goal. Instead, the technologies developed for such systems can and should be applied to secure custom or special-purpose tasks as part of a larger system, minimizing the trusted computing base required for these tasks. We observe a similar trajectory for system call interposition, which, when introduced, was envisioned as a means for sandboxing arbitrary untrusted applications [10, 11, 24]. Sandboxing complex general-purprose software proved to be difficult [9]; today, system call interposition is used fruitfully for sandboxing special-purpose processes, such as the Chromium renderer [2].

We believe that the software to be run safely on untrusted kernels will need to be designed specifically for that purpose. As we have shown, assumptions about system call behavior can be present at all levels of the software stack, not just in the application code itself. For example, we have shown that any application that calls `malloc` is potentially vulnerable. But even an application whose memory allocation has been rewritten to avoid `malloc` might find that its libraries do; for example, EGLIBC's `printf` implementation does.

If the applications that will run on untrusted kernels are written from scratch, it is no longer necessary to use the legacy Unix system call API. An important question then arises: are there other resource-management APIs that would be easier to secure as untrusted RPC interfaces? We hope to tackle this question in future work. As inspiration, we note that Mach featured a user-space memory manager [12], and that research operating systems such as the Exokernel [8] have radically reimagined the boundary between kernel and application.

Acknowledgments

We thank Dan Boneh, Nadia Heninger, David Kohlbrenner, Eric Rescorla, Stefan Savage, and Vitaly Shmatikov for helpful discussions about this work; Michael Bailey, Peter Chen, J. Alex Halderman, Peter Honeyman, Jon Oberheide, and others at the University of Michigan for their comments on an early presentation of this work; the anonymous reviewers for their helpful suggestions; and Katharine Tillman for suggesting a name for the attack.

This material is based upon work supported by National Science Foundation under grant CNS-0831532 (Cyber Trust) and by the MURI program under AFOSR Grant No. FA9550-08-1-0352.

References

[1] Anonymous. Once upon a free().... *Phrack Magazine*, 57(9), August 2001. `http://www.phrack.org/archives/57/p57_0x09_Once%20upon%20a%20free()_by_anonymous%20author.txt`.

[2] Adam Barth, Collin Jackson, Charles Reis, and The Google Chrome Team. The security architecture of the Chromium browser. Online: `http://seclab.stanford.edu/websec/chromium/`, 2008.

[3] blackngel. Malloc des-maleficarum. *Phrack Magazine*, 66(10), November 2009. `http://www.phrack.org/archives/66/p66_0x0a_Malloc%20Des-Maleficarum_by_blackngel.txt`.

[4] blackngel. ptmalloc v2 & v3: Analysis & corruption. *Phrack Magazine*, 67(8), November 2010. `http://www.phrack.org/archives/67/p67_0x08_The%20House%20Of%20Lore:%20Reloaded%20ptmalloc%20v2%20&%20v3:%20Analysis%20&%20Corruption_by_blackngel.txt`.

[5] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide

long-lasting security? The case of return-oriented programming and the AVC Advantage. In David Jefferson, Joseph Lorenzo Hall, and Tal Moran, editors, *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, August 2009.

[6] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In James Larus, editor, *Proceedings of ASPLOS 2008*, pages 2–13. ACM Press, March 2008.

[7] ISO/IEC FDIS 9899:1999 (E). *Programming languages – C*. ISO, 1999.

[8] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An operating system architecture for application-level resource management,. In Mark Weiser, editor, *Proceedings of SOSP 1995*, pages 251–66. ACM Press, December 1995.

[9] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In Virgil Gligor and Mike Reiter, editors, *Proceedings of NDSS 2003*. Internet Society, February 2003.

[10] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In Mike Reiter and Dan Boneh, editors, *Proceedings of NDSS 2004*. Internet Society, February 2004.

[11] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In Greg Rose, editor, *Proceedings of USENIX Security 1996*. USENIX, July 1996.

[12] David B. Golub and Richard P. Draves. Moving the default memory manager out of the mach kernel. In Alan Langerman, editor, *Proceedings of Mach Symposium 1991*, pages 177–88, November, 1991. USENIX.

[13] Michel Kaempf. Vudo malloc tricks. *Phrack Magazine*, 57(8), August 2001.
`http://www.phrack.org/archives/57/p57_0x08_Vudo%20malloc%20tricks_by_MaXX.txt`.

[14] David Lie, Chandramohan Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In Larry Peterson, editor, *Proceedings of SOSP 2003*, pages 178–92. ACM Press, October 2003. XOMJJ

[15] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. Minimal tcb code execution (extended abstract). In Birgit Pfitzmann and Patrick McDaniel, editors, *Proceedings of IEEE Security & Privacy ("Oakland") 2007*, pages 267–72. IEEE Computer Society, May 2007.

[16] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. How low can you go? Recommendations for hardware-supported minimal TCB code execution. In James Larus, editor, *Proceedings of ASPLOS 2008*, pages 14–25. ACM Press, March 2008.

[17] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. Flicker: An execution infrastructure for TCB minimization. In Steven Hand, editor, *Proceedings of EuroSys 2008*, pages 315–28. ACM Press, March 2008.

[18] Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter. Safe passage for passwords and other sensitive data. In Giovanni Vigna, editor, *Proceedings of NDSS 2009*. The Internet Society, February 2009.

[19] Gene Novark and Emery D. Berger. DieHarder: Securing the heap. In Angelos D. Keromytis and Vitaly Shmatikov, editors, *Proceedings of CCS 2010*. ACM Press, October 2010.

[20] Jon Oberheide. The stack is back. Presented at Infiltrate 2012, January 2012. Presentation. Slides:
`http://jon.oberheide.org/files/infiltrate12-thestackisback.pdf`.

[21] Phantasmal Phantasmagoria. The malloc maleficarum: Glibc malloc exploitation techniques. Bugtraq, October 2005.
`http://seclists.org/bugtraq/2005/Oct/118`.

[22] Dan R.K. Ports and Tal Garfinkel. Towards application security on untrusted operating systems. In Niels Provos, editor, *Proceedings of HotSec 2008*. USENIX, July 2008.

[23] POSIX.1-2008/IEEE Std 1003.1-2008. *The Open Group Base Specifications Issue 7*. IEEE and The Open Group, 2008.

[24] Niels Provos. Improving host security with system call policies. In Vern Paxson, editor, *Proceedings of USENIX Security 2003*. USENIX, August 2003.

[25] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2000.

[26] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In Wenke Lee, editor, *Proceedings of NDSS 2003*. Internet Society, February 2003.

[27] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *Trans. Info. & Sys. Sec.*, 2012. To appear.

[28] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections in Windows Vista. Presented at Black Hat 2008, August 2008. Online:
`http://www.phreedom.org/research/bypassing-browser-memory-protections/bypassing-browser-memory-protections.pdf`.

[29] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In Anja Feldmann and Laurent Mathy, editors, *Proceedings of IMC 2009*, pages 15–27. ACM Press, November 2009.