Efficient Techniques for Comprehensive Protection from Memory Error Exploits

Sandeep Bhatkar, R. Sekar and Daniel C. DuVarney
Department of Computer Science,
Stony Brook University, Stony Brook, NY 11794
{sbhatkar,sekar,dand}@cs.sunysb.edu

Abstract

Despite the wide publicity received by buffer overflow attacks, the vast majority of today's security vulnerabilities continue to be caused by memory errors, with a significant shift away from stack-smashing exploits to newer attacks such as heap overflows, integer overflows, and format-string attacks. While comprehensive solutions have been developed to handle memory errors, these solutions suffer from one or more of the following problems: high overheads (often exceeding 100%), incompatibility with legacy C code, and changes to the memory model to use garbage collection. *Address space randomization (ASR)* is a technique that avoids these drawbacks, but existing techniques for ASR do not offer a level of protection comparable to the above techniques. In particular, attacks that exploit relative distances between memory objects aren't tackled by existing techniques. Moreover, these techniques are susceptible to information leakage and brute-force attacks. To overcome these limitations, we develop a new approach in this paper that supports comprehensive randomization, whereby the absolute locations of all (code and data) objects, as well as their relative distances are randomized. We argue that this approach provides probabilistic protection against all memory error exploits, whether they be known or novel. Our approach is implemented as a fully automatic source-to-source transformation which is compatible with legacy C code. The address-space randomizations take place at load-time or runtime, so the same copy of the binaries can be distributed to everyone — this ensures compatibility with today's software distribution model. Experimental results demonstrate an average runtime overhead of about 11%.

1 Introduction

A vast majority of security vulnerabilities reported in recent years have been based on memory errors in C (and C++) programs. In the past two years, the CERT Coordination Center (now US-CERT) [5] has issued about 54 distinct advisories involving COTS software, of which 44 (over 80%) are due to memory errors. In spite of the wide publicity received by buffer overflow attacks, the fraction of vulnerabilities attributed to memory errors has grown steadily in the past ten years or so.

Even as techniques such as "stack-guarding" [10] have been developed to defeat the most common form of exploit, namely stack-smashing, newer forms of attacks continue to be discovered. The fraction of memory error exploits attributed to newer forms of attacks such as heap overflows, integer overflows, and format-string attacks have increased significantly in the past two years: 22 of the 44 CERT/CC advisories in the past two years are attributed to these newer forms of attacks, as opposed to 32 that were attributed to stack-smashing. (Note that some advisories report multiple vulnerabilities together.) This spate of new memory-related attacks suggests that new ways to exploit memory errors will continue to be discovered, and hence these errors will likely to be the principal source of cyber attacks in the foreseeable future.

We can, once for all, eliminate this seemingly endless source of vulnerabilities by adding complete memory error protection. Unfortunately, existing techniques such as backwards-compatible bounds checking [18] and its descendant CRED [28] are associated with high overheads, sometimes exceeding 1000%. Lower overheads are reported in [34], but the overheads can still be over 100% for some programs. Approaches such as CCured [24] and Cyclone [17] can bring down this overhead significantly, but aren't compatible with legacy C code. Nontrivial programming effort is often required to port existing C programs so that they can work with these tools. Precompiled libraries can pose additional compatibility problems. Finally, these two approaches rely on garbage collection instead of the explicit memory management model used in C programs, which can pose another obstacle to their widespread acceptance.

Whereas the above approaches are concerned with preventing *all invalid memory accesses,* we present an approach with a more limited goal: *it only seeks to ensure that the results of any invalid access are unpredictable.* We show that this goal can be achieved with a much lower runtime overhead of about 10%. Our approach avoids the compatibility issues mentioned above with complete memory error protection techniques. Although the protection provided by our approach is only probabilistic, we show that for all known classes of attacks, the odds of success are very small.

Our approach is based on the concept of address obfuscation [4], whose goal is to obscure the location of code and data objects that are resident in memory. Several techniques have been developed to achieve such obfuscation using randomization techniques [14, 25, 4, 33]. Although these techniques can provide protection against most known types of memory error exploits, they are vulnerable to several classes of attacks including relative-address attacks, information leakage attacks, and attacks on randomization [29]. More importantly, they do not provide systematic protection against all memory error exploits, which means that other attacks on these techniques will likely continue to be discovered in the future. In contrast, the approach developed in this paper is aimed at protecting against all memory error exploits, whether they be known or unknown.

1.1 Overview of Approach

Our approach makes the memory locations of program objects (including code as well as data objects) unpredictable. This is achieved by randomizing the absolute locations of all objects, as well as the relative distance between any two objects.

We adopted a program transformation based approach to perform the randomizations. Ideally, the transformations can be applied on program binaries. However, due to the difficulties associated with analysis of binaries, we need additional information to be present in the binaries in order to make the transformations feasible [12]. This requires changes to the system tools such as compilers and linkers, which are used to generate binaries. Our current implementation avoids this option, and instead uses a source-to-source transformation on C programs. Note that a particular randomization isn't hard-coded into the transformed code. Instead, the transformation produces a *self-randomizing program:* a program that randomizes itself each time it is run, or continuously during runtime. This means that the use of our approach doesn't, in any way, change the software distribution model that is prevalent today. Software providers can continue to distribute identical copies of program binaries to all users.

In our approach, the location of code objects is randomized using binary transformation at load-time. Static data objects are randomized at the beginning of program execution. Stack objects are continuously randomized throughout runtime. The key techniques used in achieving this randomization are outlined below.

- *Randomizing stack-resident variables.* Our approach randomizes the locations of stack-allocated variables continuously at runtime. It is based on:
  - *Shadow stack for buffer-type variables.* A separate stack is used for allocating arrays, as well as structures whose addresses are taken. By separating buffer-type variables, any overflow attacks are prevented from corrupting information such as return address or local variables that have pointer types. Moreover, to randomize the effect of overflows from one buffer-type variable to the next, we randomize the order of allocation of these buffer variables in a different way for each call.
  - *Randomizing the base of activation records.* To obscure the location of other stack-resident data, we randomize the base of the stack, as well as introduce random-sized gaps between successive stack frames.
- *Randomizing static data.* The location of each static variable, as well the relative order of these variables, is determined at the start of execution of the transformed program. Our transformation converts every access to a static variable to use an additional level of indirection, e.g., an access `v` is converted into something like `(*v_ptr)`. At the beginning of program execution, the location of the variable `v` is determined, and this value is stored in `v_ptr`. Note that, in effect, the only static variables left in the transformed program are the pointer variables such as `v_ptr`. Although these variables have predictable locations, attacks on them are prevented by storing them in read-only memory.
- *Randomizing code.* Code is randomized at the granularity of individual functions. Our technique associates a function pointer `f_ptr` with each function `f`, and transforms every call into

an indirect call using `f_ptr`. The order of different functions can now be freely permuted in the binary, as long as `f_ptr` is updated to reflect the new location of the function body for `f`. Although the location of `f_ptr` variables are predictable, attacks on them are prevented by write-protecting them.

In addition to these steps, our approach randomizes the base of the heap, gaps between heap allocations, and the location of functions in shared libraries.

1.2 Impact of Comprehensive Randomization on Memory Error Exploits

Intuitively, a memory error occurs in C programs when the object accessed via a pointer expression is different from the one intended by the programmer. The intended object is called the *referent* of the pointer. Memory errors can be classified into *spatial* and *temporal errors:*

I. *A spatial error* occurs when dereferencing a pointer that is outside the bounds of its referent. It may be caused as a result of:

(a) *Dereferencing non-pointer data,* e.g., a pointer may be (incorrectly) assigned from an integer, and dereferenced subsequently. Our randomization makes the result of this dereferencing unpredictable. The same integer value, when interpreted as a pointer, will reference different variables (or code) for each execution of the program.

(b) *Dereferencing uninitialized pointers.* This case differs from the first case only when a memory object is reallocated. In the absence of our transformation, the contents of uninitialized pointers may become predictable if the previous use of the same memory location can be identified. For instance, suppose that during an invocation of a function `f`, its local variable `v` holds a valid pointer value. If `f` is invoked immediately by its caller, then `v` will continue to contain the same valid pointer even before its initialization. By introducing random gaps in the stack, our approach changes the location of `v` across different invocations of `f`, thereby making the result of uninitialized pointer dereferences unpredictable. A similar argument applies to reallocation within the heap, as our transformation introduces random-sized gaps between heap objects.

(c) *Valid pointers used with invalid pointer arithmetic.* The most common form of memory access error, namely, out-of-bounds array access, falls in this category. Since the relative distances between memory objects are randomized in our approach, one cannot determine the target object that will be accessed as a result of invalid pointer arithmetic.

II. *A temporal error* occurs when dereferencing a pointer whose referent no longer exists, i.e., it has been freed previously. If the invalid access goes to an object in free memory, then it causes no errors. But if the memory has been reallocated, then temporal errors allow the contents of the reallocated object to be corrupted using the invalid pointer. Note that this case is essentially the same as case I(b), in that the results of such errors become predictable only when the purpose of reuse of the memory location is predictable. Since our transformation makes this unpredictable, there is no way for attackers to predict the result of memory dereferences involving temporal errors.

It may appear that temporal errors, and errors involving uninitialized pointers, are an unlikely target for attackers. In general, it may be hard to exploit such errors if they involve heap objects, as heap allocations tend to be somewhat unpredictable even in the absence of any randomizations. However, stack allocations are highly predictable, so these errors can be exploited in attacks involving stack-allocated variables. Our randomization technique reduces this likelihood.

We point out that previous techniques for ASR address case I(a), but not the other three cases, and hence the approach presented in this paper is the first randomization technique that has the potential to defend against all memory exploits.

1.3 Benefits of Our Approach

Our approach provides the following benefits:

- *Ease of use.* Our approach is implemented as an automatic, source-to-source transformation, and is fully compatible with legacy C code. It can interoperate with preexisting (untransformed) libraries. Finally, it doesn't change the current model of distributing identical copies of software (on CDs or via downloads) to all users.
- *Comprehensive randomization.* At runtime, the absolute as well as relative distances between all memory-resident objects are randomized. Hence the approach presented in this paper can address the full range of attacks that exploit memory errors. This contrasts with previous ASR approaches [25, 4] that are vulnerable to relative-address-dependent attacks.
- *Portability across multiple platforms.* The vast majority of our randomizations are OS and archi-

tecture independent. This factor eases portability of our approach to different platforms. Of particular significance is the fact that our approach sidesteps the binary disassembly and rewriting problems that have proven to be the Achilles' heel of other techniques that attempt transformations or randomization of binary code.

- *Low runtime overhead.* Our approach produces low overheads, typically in the range of 10%. It is interesting to note that, in spite of providing much more comprehensive randomization, our overheads are comparable to that of [4, 25].
- *Ease of deployment.* Our approach can be applied to individual applications *without requiring changes* to the OS kernel, system libraries or the software distribution models. It empowers code producers and code consumers to improve security of individual applications without requiring cooperation of the OS vendors. This ability to deploy at an application granularity provides an incremental deployment path, where computers can gradually become more robust against memory error exploits even when their operating systems aren't upgraded for years.

1.4 Paper Organization

The rest of the paper is organized as follows. In Section 2, we describe transformations to introduce various randomizations. Section 3 describes our implementation of these transformations. Runtime overheads introduced by our approach are discussed in Section 4. Section 5 discusses the effectiveness of our approach against different attacks, and analyzes the probability of mounting successful attacks. Related work is covered in Section 6. Finally, concluding remarks appear in Section 7.

2 Transformation Approach

2.1 Static Data Transformations

One possible approach to randomize the location of static data is to recompile the data into position-independent code (PIC). This is the approach taken in PaX ASLR [25], as well as in [4]. A drawback of this approach is that it does not protect against relative address attacks, e.g., an attack that overflows past the end of a buffer to corrupt security-critical data that is close to the buffer. Moreover, an approach that relies only on changes to the base address is very vulnerable to information leakage attacks, where an attacker may mount a successful attack just by knowing the address of any static variable, or the base address of the static area. Finally, on operating systems such as Linux, the base address of different memory sections for any process is visible to any user with access to that system, and hence the approach does not offer much protection from this class of attacks.

For the reasons described above, our approach is based on permuting the order of static variables at the beginning of program execution. In particular, for each static variable `v`, an associated (static) pointer variable `v_ptr` is introduced in the transformed program. All accesses to the variable `v` are changed to reference `(*v_ptr)` in the transformed program. Thus, the only static variables in the transformed program are these `v_ptr` variables, and the program no longer makes any reference to the original variable names such as `v`.

At the beginning of program execution, control is transferred to an initialization function introduced into the transformed program. This function first allocates a new region of memory to store the original static variables. This memory is allocated dynamically so that its base address can be chosen randomly. Next, each static variable `v` in the original program is allocated storage within this region, and `v_ptr` is updated to point to the base of this storage.

To permute the order of variables, we proceed as follows. If there are n static variables, a random number generator is used to generate a number i between 1 and n. Now, the ith variable is allocated first in the newly allocated region. Now, there are $n-1$ variables left, and one can repeat the process by generating a random number between 1 and $n-1$ and so on.

Note that bounds-checking errors dominate among memory errors. Such errors occur either due to the use of an array subscript that is outside its bounds, or more generally, due to incorrect pointer arithmetic. For this reason, our transformation separates buffer-type variables, which can be sources of bounds-checking errors, from other types of variables. Buffer-type variables include all arrays and structures containing arrays. In addition, they include any variable whose address is taken, since it may be used in pointer arithmetic, which can in turn lead to out-of-bounds access.

All buffer-type variables are allocated separately from other variables. Inaccessible memory pages (neither readable nor writable) are introduced before and after the memory region containing buffer variables, so that any buffer overflows from these variables cannot corrupt non-buffer variables. The order of buffer-type variables is randomized as mentioned above. In addition, inaccessible pages are also introduced periodically within this region to limit the scope of buffer-to-buffer overflows.

Finally, all of the `v_ptr` variables are write-protected. Note that the locations of these variables are predictable, but this cannot be used as a basis for attacks due to write-protection.

2.2 Code Transformations

As with static data, one way to randomize code location is to generate PIC code, and map this at a randomly chosen location at runtime. But this approach has several drawbacks as mentioned before, so our approach involves randomizing at a much finer granularity. Specifically, our randomization technique works at the granularity of functions. To achieve this, a function pointer `f_ptr` is associated with each function `f`. It is initialized with the value of `f`. All references to `f` are replaced by `(*f_ptr)`.

The above transformation avoids calls using absolute addresses, thereby laying the foundation for relocating function bodies in the binary. But this is not enough: there may still be jumps to absolute addresses in the code. With C-compilers, such absolute jumps are introduced while translating switch statements. In particular, there may be a jump to location `jumpTable[i]`, where `i` is the value of the switch expression, and `jumpTable` is a constant table constructed by the compiler. The `i`th element of this table contains the address of the corresponding case of the switch statement. To avoid absolute address dependency introduced in this translation, we transform a switch into a combination of if-then-else and goto statements. Efficient lookup of case values can be implemented using binary search, which will have $O(log\ N)$ time complexity. However, in our current implementation we use sequential search. In theory, this transformation can lead to decreased performance, but we have not seen any significant effect due to this change in most programs.

On a binary, the following actions are performed to do the actual randomization. The entire code from the executable is read. In addition, the location of functions referenced by each `f_ptr` variable is read from the executable. Next, these functions are reordered in a random manner, using a procedure similar to that used for randomizing the order of static variables. Random gaps and inaccessible pages are inserted periodically during this process in order to introduce further uncertainty in code locations, and to provide additional protection. The transformation ensures that these gaps do not increase the overall space usage for the executable by more than a specified parameter (which has the value of 100% in our current implementation). This limit can be exceeded if the original code size is smaller than a threshold (32K).

After relocating functions, the initializations of `f_ptr` variables are changed so as to reflect the new location of each function. The transformed binary can then be written back to the disk. Alternatively, the transformation could be done at load-time, but we have not implemented this option so far.

It is well known that binary analysis and transformation are very hard problems [26]. To ease this problem, our transformation embeds "marker" elements, such as an array of integers with predefined values, to surround the function pointer table. These markers allow us to quickly identify the table and perform the above transformation, without having to rely on binary disassembly.

As a final step, the function pointer table needs to be write-protected.

2.3 Stack Transformations

To change the base address of the stack, our transformation adds initialization code that subtracts a large random number (of the order of 10^8) from the stack pointer. In addition, all of the environment variables and command line arguments are copied over, and the original contents erased to avoid leaving any data that may be useful to attackers (such as file names) at predictable locations. Finally, the contents of the stack above the current stack pointer value are write-protected. (An alternative to this approach is to directly modify the base address of the stack, but this would require changes to the OS kernel, which we want to avoid. For instance, on Linux, this requires changes to `execve` implementation.)

The above transformation changes the absolute locations of stack-resident objects, but has no effect on relative distances between objects. One possible approach to randomize relative distances is to introduce an additional level of indirection, as was done for static variables. However, this approach will introduce high overheads for each function call. Therefore we apply this approach only for buffer-type local variables. (Recall that buffer-type variables also include those whose address is explicitly or implicitly used in the program.) Specifically, for each buffer-type variable, we introduce a pointer variable to point to it, and then allocate the buffer itself on a second stack called the *shadow stack*. Consider a local variable declaration `char buf[100]` within a function, `func`. This variable can be replaced by a pointer with the following definition:

```
char (*buf_ptr)[100]
```

On entry of `func`, memory for `buf` is allocated using:

```
    buf_ptr = shadow_alloc(sizeof(char [100]))
```

Allocations of multiple buffers are performed in a random order similar to static variables. Also, the allocator function allocates extra memory of a random size (currently limited to a maximum of 30%) between buffers, thereby creating random gaps between adjacent buffers. Finally, all occurrences of `buf` in the body of `func` are replaced with `(*buf_ptr)`.

Our transformation does not change the way other types of local variables are allocated, so they get allocated in the same order. However, since the addresses of these variables never get taken, they cannot be involved in attacks that exploit knowledge of relative distances between variables. In particular, stack-smashing attacks become impossible, as the return address is on the regular stack, whereas the buffer overflows can only corrupt the shadow stack. In addition, attacks using absolute addresses of stack variables do not work, as the absolute addresses are randomized by the (random) change to the base address of the stack.

Note that function parameters may be buffer-type variables. To eliminate the risk of overflowing them, we copy all buffer-type parameters into local variables, and use only the local variables from there on. Buffer type parameters are never accessed in code, so there is no possibility of memory errors involving them. (An alternative to this approach is to ensure that no buffer-type variables are passed by value. But this requires the caller and callee code to be transformed simultaneously, thereby potentially breaking separate compilation.)

As a final form of stack randomization, we introduce random gaps between stack frames. This makes it difficult to correlate the locations of local variables across function invocations, thereby randomizing the effect of uninitialized pointer access and other temporal errors. Before each function call, code is added to decrements stack pointer by a small random value. After the function call, this padding is removed. The padding size is a random number generated at runtime, so it will vary for each function invocation.

2.4 Heap Transformations

To modify the base address of the heap, code is added to make a request for a large data block before the first heap allocation request is made. The details of this step will vary with the underlying `malloc` implementation, and are described later on.

To randomize the relative distance between heap objects, calls to `malloc()` are intercepted by a wrapper function, and the size of the request increased by a random amount, currently between 0% and 30%.

Additional randomizations are possible as well. For instance, we can intercept calls to `free`, so that some of the freed memory is not passed on to malloc, but simply result in putting the the buffer in a temporary buffer. The implementation of the malloc wrapper can be modified to perform allocations from this buffer, instead of passing on the request to `malloc`. Since heap objects tend to exhibit a significant degree of randomness naturally, we have not experimented with this transformation.

2.5 DLL Transformations

Ideally, DLLs should be handled in the same way as executable code: the order of functions should be randomized, and the order of static variables within the libraries should be randomized. However, DLLs are shared across multiple programs. Randomization at the granularity of functions, if performed at load time on DLLs, will create *copies* of these DLLs, and thus rule out sharing. To enable sharing, randomization can be performed on the disk image of the library rather than at load time. Such randomization has to be performed periodically, e.g., at every restart of the system.

A second potential issue with DLLs is that their source code may not be available. In this case, the base address of the DLL can be randomized in a manner similar to [25, 4]. However, this approach does not provide sufficient range of randomization on 32-bit architectures. In particular, with a page size of 4096 ($= 2^{12}$) bytes on Linux, uncertainty in the base address of a library cannot be much larger than 2^{16}, which makes them susceptible to brute-force attacks [29]. We address this problem by a link-time transformation to prepend each DLL with junk code of random size between 0 and page size. The size of this junk code must be a multiple of 4, so this approach increases the space of randomization to $2^{16} * 2^{12}/4 = 2^{26}$.

2.6 Other Randomizations

Randomization of PLT and GOT. In a dynamically linked ELF executable, calls to shared library functions are resolved at runtime by the dynamic linker. The GOT (global offset table) and PLT (procedure linkage table) play crucial roles in resolution of library functions. The GOT stores the addresses of external functions, and is part of the data segment. The PLT, which is part of the code segment, contains entries that call addresses stored in the GOT.

From the point of view of an attacker looking to access system functions such as `execve`, the PLT and GOT provide "one-stop shopping," by conveniently collecting together the memory locations of all system functions in one place. For this reason, they have become a common target for attacks. For instance,

- if an attacker knows the absolute location of the PLT, then she can determine the location within the PLT that corresponds to the external function `execve`, and use this address to overwrite a return address in a stack-smashing attack. Note that this attack works even if the locations of all functions in the executable and libraries have been randomized

- if an attacker knows the absolute location of the GOT, she can calculate the location corresponding to a commonly used function such as the `read` system call, and overwrite it with a pointer to attack code injected by her. This would result in the execution of attack code when the program performs a `read`.

It is therefore necessary to randomize the locations of the PLT and GOT, as well as the relative order of entries in these tables. However, since the GOT and PLT are generated at link-time, we cannot control them using source code transformation. One approach for protecting the GOT is to use the eager linking option, and then write-protect it at the beginning of the `main` program. An alternative approach that uses lazy linking (which is the default on Linux) is presented in [33].

The main complication in relocating the PLT is to ensure that any references in the program code to PLT entries be relocated. Normally, this can be very difficult, because there is no way to determine through a static analysis of a binary whether a constant value appearing in the code refers to a function, or is simply an integer constant. However, our transformation has already addressed this problem: every call to an entry `e` in the PLT will actually be made using a function pointer `e_ptr` in the transformed code. As a result, we treat each entry in the PLT as if it is a function, and relocate it freely, as long as the `e_ptr` is correctly updated.

Randomization of read-only data. The read-only data section of a program's executable consists of constant variables and arrays whose contents are guaranteed not to change when the program is being run. Attacks which corrupt data cannot harm read-only data. However, if their location is predictable, then they may be used in some attacks that need meaningful argument values, e.g., a typical return-to-libc attack will modify a return address on the stack to point to `execve`, and put pointer arguments to `execve` on the stack. For this attack to succeed, an attacker has to know the absolute location of a string constant such as `/bin/bash` which may exist in the read-only section.

Note that our approach already makes return-to-libc attacks very difficult. Nevertheless, it is possible to make it even more difficult by randomizing the location of potential arguments in such attacks. This can be done by introducing variables in the program to hold constant values, and then using the variables as arguments instead of the constants directly. When this is done, our approach will automatically relocate these constants.

3 Implementation

The main component of our implementation is a source code transformer which uses CIL [23] as the front-end, and Objective Caml as the implementation language. CIL translates C code into a high-level intermediate form which can be transformed and then emitted as C source code, considerably facilitating the implementation of our transformation.

Our implementation also includes a small platform-specific component that supports transformations involving code and DLLs.

The implementation of these components are described in greater detail below. Although the source-code transformation is fairly easy to port to different OSes, the description below refers specifically to our implementation on an x86/Linux system.

3.1 Static Data Transformations

Static data can be initialized or uninitialized. In an ELF executable, the initialized data is stored in the `.data` section, and the uninitialized data is stored in the `.bss` section. For uninitialized data, there is no physical space required in the executable. Instead, the executable only records the total size of the `.bss` section. At load-time, the specified amount of memory is allocated and initialized with zeroes.

In the transformed program, initializations have to be performed explicitly in the code. First, all newly introduced pointer variables should be initialized to point to the locations allocated to hold the values of the original static variables. Next, these variables need to be initialized. We illustrate these transformations through an example:

```
int a = 1;
char b[100];
extern int c;

void f() {
    while (a < 100) b[a] = a++;
}
```

We transform the above declarations, and also add an initialization function to allocate memory for the variables defined in the source file as shown below:

```
int *a_ptr;
char (*b_ptr) [100];
extern int *c_ptr;

void __attribute__ ((constructor)) data_init(){

    struct {
      void *ptr;
      unsigned int size;
      BOOL is_buffer;
    } alloc_info[2];

    alloc_info[0].ptr = (void *) &a_ptr;
    alloc_info[0].size = sizeof(int);
    alloc_info[0].is_buffer = FALSE;
    alloc_info[1].ptr = (void *) &b_ptr;
    alloc_info[1].size = sizeof(char [100]);
    alloc_info[1].is_buffer = TRUE;

    static_alloc(alloc_info, 2);

    (*a_ptr) = 1;
}

void f() {
      while ((*a_ptr) < 100)
        (*b_ptr)[(*a_ptr)] = (*a_ptr)++;
}
```

For the initialization function `data_init()`, we use `constructor` attribute so that it is invoked automatically before execution enters `main()`. Each element in the array `alloc_info` stores information about a single static variable, including the location of its pointer variable, its size, etc. Memory allocation is done by the function `static_alloc`, which works as follows. First, it allocates the required amount of memory by using a `mmap`. (Note that `mmap` allows its caller to specify the start address and length of a segment, and this capability is used to randomize the base address of static variables.) Second, it randomly permutes the order of static variables specified in `alloc_info`, and introduces gaps and protected memory sections in-between some variables. Finally, it zeroes out the memory allocated to static variables. After the call to `static_alloc`, code is added to initialize those static variables that are explicitly initialized.

Other than the initialization step, the rest of the transformation is very simple: replace the occurrence of each static variable to use its associated pointer variable, i.e., replace occurrence of `v` by `(*v_ptr)`.

The data segment might contain other sections included by the static linker. Of these sections, `.ctors`, `.dtors` and `.got` contain code pointers. Therefore we need to protect these sections, or otherwise attackers can corrupt them to hijack program control. The sections `.dtors` and `.ctors`, which contain global constructors and destructors, can be put into a read-only segment by changing a linker script option.

Section `.got` contains GOT, whose randomization was discussed in the previous section in the context of randomization of PLT (See Section 2.6).

All of the `v_ptr` variables are write-protected by initialization code that is introduced into `main`. This code first figures out the boundaries of the data segment, and then uses the `mprotect` system call to apply the write protection.

3.2 Code Transformations

Code transformation mainly involves converting direct function calls into indirect ones. We store function pointers in an array, and dereference elements from this array to make the function calls. The details can be understood with an example. Consider a source file containing following piece of code:

```
char *f();
void g(int a) { ... }
void h() {
  char *str;
  char *(*fptr)();
  ...
  fptr = &f;
  str = (*fptr)();
  g(10);
}
```

The above code will be transformed as follows:

```
void *const func_ptrs[] =
  {M1, M2, M3, M4, (void *)&f, (void *)&g,
   M5, M6, M7, M8};

char *f();
void g(int a) { ... }
void h() {
  char *str;
  char *(*fptr)();
  ...
  fptr = (char *(*)())func_ptrs[4];
  str = (*fptr)();
  (*((void (*)(int)) (func_ptrs[5])))(10);
}
```

The function pointer array in each source file contains locations of functions used in that file. Due to the `const` modifier, the array becomes part of the `.rodata` section in the code segment of the corresponding ELF executable, and is hence write-protected.

The `func_ptrs` array is bounded on each end with a distinctive, 128-bit pattern that is recorded in the marker variables `M1` through `M8`. This pattern is assumed to be unique in the binary, and can be easily identified when scanning the binary. These markers simplify binary transformations, as we no longer need to disassemble the binary for the purpose of function-reordering transformation. Instead, the original locations of functions can be identified from the contents of this array. By sorting the array elements, we can identify the beginning as well as the end of each function. (The end of a function is assumed to just precede the beginning of the next function in the sorted array.) Now, the binary transformation simply needs to randomly reorder function bodies, and change the content of the `func_ptr` array to point to these new locations. We adapted the LEEL binary-editing tool [35] for performing this code transformation.

In our current implementation, we do not reorder functions at load time. Instead, the same effect is achieved by modifying the executable periodically.

3.3 Stack Transformations

In our current implementation, the base of the stack is randomized by decrementing a large number from the stack pointer value. This is done in the `__libc_start_main` routine, and hence happens before the invocation of `main`. Other stack-related transformations are implemented using a source-code transformation. Transformation of buffer-type local variables is performed in a manner similar to that of static variables. The only difference is that their memory is allocated on the shadow stack.

Introduction of random-sized gaps between stack frames is performed using the `alloca` function, which is converted into inline assembly code by `gcc`. There are two choices on where this function is invoked: (a) immediately before calling a function, (b) immediately after calling a function, i.e., at the beginning of the called function. Note that option (b) is weaker than option (a) in a case where a function f is called repeatedly within a loop. With (a), the beginning of the stack frame will differ for each call of f. With (b), all calls to f made within this loop will have the same base address. Nevertheless, our implementation uses option (b), as it works better with some of the compiler optimizations.

Handling setjmp/longjmp. The implementation of shadow stack needs to consider subroutines such as `setjmp()` and `longjmp()`. A call to `setjmp()` stores the program context which mainly includes the stack pointer, the frame pointer and the program counter. A subsequent call to `longjmp()` restores the program context and the control is transferred to the location of the `setjmp()` call. To reflect the change in the program context, the shadow stack needs to be modified. Specifically, the top of shadow stack needs to be adjusted to reflect the `longjmp`. This is accomplished by storing the top of the shadow stack as a local variable in the regular stack and restoring it at the point of function return. As a result, the top of shadow stack will be properly positioned before the first allocation following the `longjmp`. (Note that we do not need to change the implementation of `setjmp` or `longjmp`.)

3.4 Heap Transformations

Heap-related transformations may have to be implemented differently, depending on how the underlying heap is implemented. For instance, suppose that a heap implementation allocates as much as twice the requested memory size. In this case, randomly increasing a request by 30% will not have much effect on many memory allocation requests. Thus, some aspects of randomization have to be matched to the underlying heap implementation.

For randomizing the base of heap, we could make a dummy `malloc()` call at the beginning of program execution, requesting a big chunk of memory. However, this would not work for `malloc()` as implemented in GNU `libc`: for any chunk larger than 4K, GNU `malloc` returns a separate memory region created using the `mmap` system call, and hence this request doesn't have any impact on the locations returned by subsequent `malloc`'s.

We note that `malloc` uses the `brk` system call to allocate heap memory. This call simply changes the end of the data segment. Subsequent requests to `malloc` are allocated from the newly extended region of memory. In our implementation, a call to `brk` is made before any `malloc` request is processed. As a result, locations returned by subsequent `malloc` requests will be changed by the amount of memory requested by the previous `brk`. The length of the extension is a random number between 0 and 10^8. The extended memory is write-protected using the `mprotect` system call.

In addition, each `malloc` request is increased by a random factor as described earlier. This change is performed in a wrapper to `malloc` that is incorporated in the modified C library used by our implementation.

Program	Workload
`Apache-1.3.33`	Webstone 2.5, client connected over 100Mbps network.
`sshd-OpenSSH_3.5p1`	Run a set of commands from ssh client.
`wu-ftpd-2.8.0`	Run a set of different ftp commands.
`bison-1.35`	Parse C++ grammar file.
`grep-2.0`	Search a pattern in files of combined size 108MB.
`bc-1.06`	Find factorial of 600.
`tar-1.12`	Create a tar file of a directory of size 141MB.
`patch-2.5.4`	Apply a 2MB patch-file on a 9MB file.
`enscript-1.6.4`	Convert a 5.5MB text file into a postscript file.
`ctags-5.4`	Generate tag file of 6280 C source code files with total 17511 lines.
`gzip-1.2.4`	Compress a 12 MB file.

Figure 1: Test programs and workloads

3.5 DLL transformations

In our current implementation, DLL transformations are limited to changing their base addresses. Other transformations aimed at relative address randomization are not performed currently.

Base address randomization is performed at load-time and link-time. Load-time randomization has been implemented by modifying the dynamic linker `ld.so` so that it ignores the "preferred address" specified in a DLL, and maps it at a random location. Note that there is a boot-strapping problem with randomizing `ld.so` itself. To handle this problem, our implementation modifies the preferred location of `ld.so`, which is honored by the operating system loader. This approach negatively impacts the ability to share `ld.so` among executables, but this does not seem to pose a significant performance problem due to the relatively small size and infrequent use (except during process initialization) of this library.

Link-time transformation is used to address the limited range of randomization that can be achieved at load-time. In particular, the load-time addresses are limited to be multiples of page size. To provide finer granularity changes to the base address, our implementation uses the "-r" option of `ld` to generate a relocatable object file for the DLL. Periodically, the relocatable version of the DLL is linked with random-sized (between 0 and 4K bytes) junk code to produce a new DLL that is used by all programs. We envision that this relinking step will be performed periodically, or perhaps once on every system restart.

Note that this approach completely avoids distribution of source code and (expensive) recompilation of libraries. Moreover, it allows sharing of library code across multiple processes.

3.6 Other Implementation Issues

Random number generation. Across all the transformations, code for generation of random numbers is required to randomize either the base addresses or the relative distances. For efficiency, we use a pseudo-random numbers rather than cryptographically random numbers. The pseudo-random number generator is seeded with a real random number read from `/dev/urandom`.

Debugging support. Our transformation provides support for some of the most commonly used debugging features such as printing a stack trace. Note that no transformations are made to normal (i.e., non-buffer) stack variables. Symbol table information is appropriately updated after code rewriting transformations. Moreover, conventions regarding stack contents are preserved. These factors enable off-the-shelf debuggers to produce stack traces on transformed executables.

Unfortunately, it isn't easy to smoothly handle some aspects of transformation for debugging purposes. Specifically, note that accesses to global variables (and buffer-type local variables) are made using an additional level of indirection in the transformed code. A person attempting to debug a transformed program needs to be aware of this. In particular, if a line in the source code accesses a variable `v`, he should know that he needs to examine `(*v_ptr)` to get the contents of `v` in the untransformed program. Although this may seem to be a burden, we point out that our randomizing transformation is meant to be used only in the final versions of code that are shipped, and not in debugging versions.

4 Performance Results

We have collected data on the performance impact of the randomizing transformations. The transformations were divided into the following categories, and their impact studied separately.

#clients	Degradation (%)	
	Connection Rate	Response Time
2-clients	1	0
16-clients	0	0
30-clients	0	1

Figure 2: Performance overhead for Apache.

Program	Orig. CPU time	% Overheads			
		Stack	Static	Code	**All**
grep	0.33	0	0	0	**2**
tar	1.06	2	2	1	**4**
patch	0.39	2	0	0	**4**
wu-ftpd	0.98	2	0	6	**9**
bc	5.33	7	1	2	**9**
enscript	1.44	8	3	0	**10**
bison	0.65	4	0	7	**12**
gzip	2.32	6	9	4	**17**
sshd	3.77	6	10	2	**19**
ctags	9.46	10	3	8	**23**
Avg. Overhead		**5**	**3**	**3**	**11**

Figure 3: Performance overheads for benchmarks.

- **Stack**: transformations which randomize the stack base, move buffer-type variables into the shadow stack, and introduce gaps between stack frames.
- **Static data**: transformations which randomize locations of static data.
- **Code**: transformations which reorder functions.
- **All**: all of the above, plus randomizing transformations on heap and DLLs.

Figure 1 shows the test programs and their workloads. Figure 3 shows performance overheads due to each of the above categories of transformations. The original and the transformed programs were compiled using gcc 3.2.2 with -O2 optimization, and executed on a desktop running Red Hat Linux 9.0 with 1.7GHz Pentium IV processor, and 512MB RAM. Execution times were averaged over 10 runs.

For Apache server, we studied its performance separately after applying all the transformations. To measure performance of the Apache server accurately, heavy traffic from clients is required. We generated this using WebStone [32], a standard web server benchmark. We used version 2.5 of this benchmark, and ran it on a separate computer that was connected to the server through a 100Mbps network. We ran the benchmark with two, sixteen and thirty clients. In the experiments, the clients were simulated to access the web server concurrently, randomly fetching html files of size varying from 500 bytes to 5MB. The benchmark was run for a duration of 30 minutes, and the results were averaged across ten such runs. Results were finally rounded off to the nearest integral values.

Program	%age of variable accesses		
	Local		Global
	(non-buffer)	(buffer)	(static)
grep	99.9	0.004	0.1
bc	99.3	0.047	0.6
tar	96.5	0.247	3.2
patch	91.8	1.958	6.2
enscript	90.5	0.954	8.5
bison	88.2	0.400	10.9
ctags	72.9	0.186	26.9
gzip	59.2	0.018	40.7

Figure 4: Distribution of variable accesses

Program	# calls $\times 10^6$	calls/ sec. $\times 10^6$	Shadow stack allocations	
			per sec.	per call
grep	0.02	0.06	24K	0.412
tar	0.43	0.41	57K	0.140
bison	2.69	4.11	423K	0.103
bc	22.56	4.24	339K	0.080
enscript	9.62	6.68	468K	0.070
patch	3.79	9.75	166K	0.017
gzip	26.72	11.52	0K	0.000
ctags	251.63	26.60	160K	0.006

Figure 5: Calls and shadow stack allocations

We analyzed the performance impact further by studying the execution profile of the programs. For this, we instrumented programs to collect additional statistics on memory accesses made by the transformed program. Specifically, the instrumentation counts the total number of accesses made to local variables, variables on shadow stack, global variables and so on.

Figure 4 shows the dynamic profile information. (We did not consider servers in this analysis due to the difficulties involved in accurately measuring their runtimes.) From this result, we see that for most programs, the vast majority of memory accesses are to local variables. Our transformation doesn't introduce any overheads for local variables, which explains the low overheads for most programs in Figure 3. Higher overheads are reported for programs that perform a significant number of global variable accesses, where an additional memory access is necessitated by our transformation.

A second source of overhead is determined by the number of function calls made by a program. This includes the overhead due to the additional level of indirection for making function calls, the number of allocations made on shadow stack, and the introduction of inter-stack-frame gap. To analyze this overhead, we instrumented the transformed programs to collect number of function calls and number of shadow stack allocations. The results, shown in Figure 5, illustrate that programs that make a large number of function calls per second, e.g., `ctags` and `gzip` incur higher overheads. Surprisingly, `bison` also incurs high overheads despite making small number of function calls per second. So we analyzed `bison`'s code, and found that it contains several big switch statements. This could be the main reason behind the high overheads, because our current implementation performs sequential lookup for the case values. However, with binary search based implementation, we should be able to get better performance.

We point out that the profile information cannot fully explain all of the variations in overheads, since it does not take into account some of the factors involved, such as compiler optimizations and the effect of cache hits (and misses) on the additional pointer dereferences introduced in the transformed program. Nevertheless, the profile information provides a broad indication of the likely performance overheads due to each program.

5 Effectiveness

Effectiveness can be evaluated experimentally or analytically. Experimental evaluation involves running a set of well-known exploits (such as those reported on `Securityfocus.com`) against vulnerable programs, and showing that our transformation stops these exploits. We have not carried out a detailed experimental evaluation of effectiveness because today's attacks are quite limited, and do not exercise our transformation at all. In particular, they are all based on a detailed knowledge of program memory layout. We have manually verified that our transformation changes the memory locations of global variables, local variables, heap-allocated data and functions for each of the programs discussed in the previous section. It follows from this that none of the existing buffer overflow attacks will work on the transformed programs.

In contrast with the limitations of an experimental approach, an analytical approach can be based on novel attack strategies that haven't been seen before. Moreover, it can provide a measure of protection (in terms of the probability of a successful attack), rather than simply providing an "yes" or "no" answer. For this reason, we rely primarily on an analytical approach in this section. We first analyze memory error exploits in general, and then discuss attacks that are specifically targeted at randomization.

5.1 Memory Error Exploits

All known memory error exploits are based on corrupting some data in the writable memory of a process. These exploits can be further subdivided based on the attack mechanism and the attack effect. The primary attack mechanisms known today are:

- *Buffer overflows.* These can be further subdivided, based on the memory region affected: namely, *stack, heap* or *static area overflows.* We note that integer overflows also fall into this category.
- *Format string vulnerabilities.*

Attack effects can be subdivided into:

- *Non-pointer corruption.* This category includes attacks that target security-critical data, e.g., a variable holding the name of a file executed by a program.
- *Pointer corruption.* Attacks in this category are based on overwriting *data* or *code pointers.* In the former case, the overwritten value may point to *injected data* that is provided by the attacker, or *existing data* within the program memory. In the latter case, the overwritten value may correspond to *injected code* that is provided by the attacker, or *existing code* within the process memory.

Given a specific vulnerability V, the probability of its successful exploitation is given by $P(Owr) * P(Eff)$, where $P(Owr)$ denotes the probability that V can be used to overwrite a specific data item of interest to the attacker, and $P(Eff)$ denotes the probability that the overwritten data will have the effect intended by the attacker. In arriving at this formula, we make either of the following assumptions:

- (a) the program is re-randomized after each failed attack. This happens if the failure of the effect causes the victim process to crash, (say, due to a memory protection fault), and it has to be explicitly restarted.
- (b) the attacker cannot distinguish between the failure of the overwrite step from the failure of the effect. This can happen if (1) the overwrite step corrupts critical data that causes an immediate crash, making it indistinguishable from a case where target data is successfully overwritten, but has an incorrect value that causes the

program to crash, or (2) the program incorporates error-handling or defense mechanisms that explicitly masks the difference between the two steps.

Note that (a) does not hold for typical server programs that spawn children to handle requests, but (b) may hold. If neither of them hold, then the probability of a successful attack is given by $min(P(Owr), P(Eff))$.

5.1.1 *Estimating* $P(Owr)$

We estimate $P(Owr)$ separately for each attack type.

5.1.1.1 Buffer overflows

Stack buffer overflows. These overflows typically target the return address, saved base pointer or other pointer-type local variables. Note that the shadow stack transformation makes these attacks impossible, since all buffer-type variables are on the shadow stack, while the target data is on a different stack.

Attacks that corrupt one buffer-type variable by overflowing the previous one are possible, but unlikely. As shown by our implementation results, very few buffer-type variables are allocated on the stack. Moreover, it is unusual for these buffers to contain pointers (or other security-critical data) targeted by an attacker.

Static buffer overflows. As in the case of stack overflows, the likely targets are simple pointer-type variables. However, such variables have been separated by our transformation from buffer-type variables, and hence they cannot be attacked.

For attacks that use overflow from one buffer to the next, the randomization introduced by our transformation makes it difficult to predict the target that will be corrupted by the attack. Moreover, unwritable pages have been introduced periodically in-between buffer-type static variables, and these will completely rule out some overflows. To estimate the probability of successful attacks, let M denote the maximum size of a buffer overflow, and S denote the granularity at which inaccessible pages are introduced between buffer variables. Then the maximum size of a useful attack is $min(M, S)$. Let N denote the total size of memory allocated for static variables. The probability that the attack successfully overwrites a data item intended by the attacker is given by $min(M, S)/N$. With nominal values of $4KB$ for the numerator and $1MB$ for the denominator, the likelihood of success is about 0.004.

Heap overflows. In general, heap allocations are non-deterministic, so it is hard to predict the effect of overflows from one heap block to the next. This unpredictability is further increased by our transformation to randomly increase the sizes of heap allocation requests. However, there exist control data in heap blocks, and these can be more easily and reliably targeted. For instance, heap overflow attacks generally target two pointer-valued variables that are used to chain free blocks together, and appear at their beginning.

The transformation to randomly increase `malloc` requests makes it harder to predict the start address of the next heap block, or its allocation state. However, the first difficulty can be easily overcome by writing alternating copies of the target address and value many times, which ensures that the control data will be overwritten with 50% probability. We believe that the uncertainty on allocation state doesn't significantly decrease the probability of a successful attack, and hence we conclude that our randomizations do not significantly decrease $P(Owr)$. However, as discussed below, $P(Eff)$ is very low for such attacks.

5.1.1.2 Format string attacks. These attacks exploit the (obscure) `"%n"` format specifier. The specifier needs an argument that indicates the address into which the printf-family of functions will store the number of characters that have been printed. This address is specified by the attacker as part of the attack string. Note that in the transformed program, the argument corresponding to the `"%n"` format specifier will be taken from the main stack, whereas the attack string will correspond to a buffer-type variable, and be held on the shadow stack (or the heap or in a global variable). As a result, there is no way for the attacker to directly control the address into which printf-family of functions will write, and hence the usual form of format-string attack will fail.

It is possible, however, that some useful data pointers may be on the stack, and they could be used as the target of writes. The likelihood of finding such data pointers on the stack is relatively low, but even when they do exist, the inter-stack frame gaps of the order of 2^8 bytes reduces the likelihood of successful attacks to $4/2^8 = 0.016$. This factor can be further decreased by increasing the size of inter-frame gaps in functions that call printf-family of functions.

In summary, the approach described in this paper significantly reduces the success probability of most likely attack mechanisms, which include (a) overflows from stack-allocated buffers to corrupt return address or other pointer-type data on the stack, (b) overflows from static variable to another, and (c) format-string attacks. This should be contrasted with previous ASR techniques that have *no effect at all* on $P(Owr)$. Their preventive ability is based entirely on reducing $P(Eff)$ discussed in the next section.

5.1.2 *Estimating* $P(Eff)$

5.1.2.1 Corruption of non-pointer data. This class of attacks target security-critical data such as user-ids and file names used by an application. With our technique, as well as previous ASR techniques, it can be seen that $P(Eff) = 1$, as they have no bearing on the interpretation of non-pointer data. The most likely location of such security-critical data is the static area, where our approach provides protection in the form of a small $P(Owr)$. This contrasts with previous ASR approaches that provide no protection from this class of attacks.

5.1.2.2 Pointer corruption attacks.

Corruption with pointer to existing data. The probability of correctly guessing the absolute address of any data object is determined primarily by the amount of randomization in the base addresses of different data areas. This quantity can be in the range of 2^{27}, but since the objects will likely be aligned on a 4-byte boundary, the probability of successfully guessing the address of a data object is in the range of 2^{-25}.

Corruption with pointer to injected data. Guessing the address of some buffer that holds attacker-provided data is no easier than guessing the address of existing data objects. However, the odds of success can be improved by repeating the attack data many times over. If it is repeated k times, then the odds of success is given by $k \times 2^{-25}$. If we assume that the attack data is 16 bytes and the size of the overflow is limited to $4K$, then k has the value of 2^8, and $P(Eff)$ is 2^{-17}.

Corruption with pointer to existing code. The probability of correctly guessing the absolute address of any code object is determined primarily by the amount of randomization in the base addresses of different code areas. In our current implementation, the uncertainty in the locations of functions within the executable is $2^{16}/4 = 2^{14}$. We have already argued that the randomization in the base address of DLLs can be as high as 2^{-26}, so $P(Eff)$ is bounded by 2^{-14}.

This probability can be decreased by performing code randomizations at load-time. When code randomizations are performed on disk images, the amount of "gaps" introduced between functions is kept low (on the order of 64K in the above calculation), so as to avoid large increases in file sizes. When the randomization is performed in main memory, the space of randomization can be much larger, say, 128MB, thereby reducing the probability of successful attacks to 2^{-25}.

Corruption with pointer to injected code. Code can be injected only in data areas, and it does not have any alignment requirements (on x86 architectures). Therefore, the probability of guessing the address of the injected code is 2^{-27}. The attacker can increase the success probability by using a large NOP-padding before the attack code. If a padding of the order of 4KB is used, then $P(Eff)$ becomes $4K \times 2^{-27} = 2^{-15}$.

5.2 Attacks Targeting ASR

Previous ASR approaches were vulnerable to the classes of attacks described below. We describe how the approach presented in this paper fares against them.

5.2.1 *Information leakage attacks*

Programs may contain vulnerabilities that allow an attacker to "read" the memory of a victim process. For instance, the program may have a format string vulnerability such that the vulnerable code prints into a buffer that is sent back to the attacker. (Such vulnerabilities are rare, as pointed out in [29].) Armed with this vulnerability, the attacker can send a format string such as `"%x %x %x %x"`, which will print the values of 4 words near the top of the stack at the point of the vulnerability. If some of these words are known to point to specific program objects, e.g., a function in the executable, then the attacker knows the locations of these objects.

We distinguish between two kinds of information leakage vulnerabilities: *chosen pointer leakage* and *random pointer leakage*. In the former case, the attacker is able to select the object whose address is leaked. In this case, the attacker can use this address to overwrite a vulnerable pointer, thereby increasing $P(Eff)$ to 1. With random pointer leakage, the attacker knows the location of some object in memory, but not the one of interest to him. Since relative address randomization makes it impossible in general to guess the location of one memory object from the location of another memory object, random pointer leakages don't have the effect of increasing $P(Eff)$ significantly.

For both types of leakages, note that the attacker still has to successfully exploit an overflow vulnerability. The probability of success $P(Owr)$ for this stage was previously discussed.

The specific case of format-string information leakage vulnerability lies somewhere between random pointer leakage and chosen pointer leakage. Thus, the probability of mounting a successful attack based on this vulnerability is bounded by $P(Owr)$.

5.2.2 Brute force and guessing attacks

Apache and similar server programs pose a challenge for address randomization techniques, as they present an attacker with many simultaneous child processes to attack, and rapidly re-spawn processes which crash due to bad guesses by the attacker. This renders them vulnerable to attacks in which many guesses are attempted in a short period of time. In [29], these properties were exploited to successfully attack a typical Apache configuration within a few minutes. This attack doesn't work with our approach, as it relies on stack smashing. A somewhat similar attack could be mounted by exploiting some other vulnerability (e.g., heap overflow) and making repeated attempts to guess the address of some existing code. As discussed earlier, this can be done with a probability between 2^{-14} to 2^{-26}. However, the technique used in [29] for passing arguments to this code won't work with heap overflows.

5.2.3 Partial pointer overwrites

Partial pointer overwrites replace only the lower byte(s) of a pointer, effectively adding a delta to the original pointer value. These are made possible by off-by-one vulnerabilities, where the vulnerable code checks the length of the buffer, but contains an error that underestimates the size of buffer needed by 1.

These attacks are particularly effective against randomization schemes which only randomize the base address of each program segment and preserve the memory layout. By scrambling the program layout, our approach negates any advantage of a partial overwrite over a full overwrite.

6 Related Work

Runtime Guarding These techniques transform a program to prevent corruption of return addresses or other specific values. *Stackguard* [10] provides a *gcc* patch to generate code that places *canary values* around the return address at runtime, so that any overflow which overwrites the return address will also modify the canary value, enabling the overflow to be detected. *StackShield* [2] and *RAD* [7] provide similar protection, but keep a separate copy of the return address instead of using canary values. *Libsafe* and *Libverify* [2] are dynamically loaded libraries which provide protection for the return address without requiring recompilation. *ProPolice* [13] further improves these approaches to protect pointers among local variables. *FormatGuard* [8] transforms source code to provide protection from format-string attacks.

The *PointGuard* [9] approach randomizes ("encrypts") stored pointer values. It provides protection against pointer-related attacks, but not against attacks that modify non-pointer data. Moreover, the approach does not consider features of the C language, such as type casts between pointers and integers, and aliasing of pointer-valued variables with variables of other types. As a result, PointGuard may break such programs.

Runtime Bounds and Pointer Checking Several techniques [21, 1, 30, 18, 16, 19, 24, 28, 34] have been developed to prevent buffer overflows and related memory errors by checking every memory access. These techniques currently suffer from one or more of the following drawbacks: runtime overheads that can often be over 100%, incomaptibility with legacy C-code, and changes to the memory model or pointer semantics.

Compile-Time Analysis Techniques These techniques [15, 27, 31, 11, 22] analyze a program's source code to detect potential array and pointer access errors. Although useful for debugging, they are not very practical since they suffer from high false alarm rates, and often do not scale to large programs.

Randomizing Code Transformations Address randomization is an instance of the broader idea of introducing diversity in nonfunctional aspects of software, an idea suggested by Forrest, Somayaji, and Ackley [14]. Recent works have applied it to randomization of address space [25, 4, 33], operating system functions [6], and instruction sets [20, 3]. As compared to instruction set randomization, which offers protection from injected code attacks, address space randomization offers broader protection – it can defend against existing code attacks, as well as attacks that corrupt security-critical data.

Previous approaches in address space randomization were focused only on randomizing the base address of different sections of memory. In contrast, the approach developed in this paper implements randomization at a much finer granularity, achieving relative as well as absolute address randomization. Moreover, it makes certain types of buffer overflows impossible. Interestingly, our implementation can achieve all of this, while incurring overheads that are about the same as the previous techniques [4].

7 Conclusion

Address space randomization (ASR) is an technique which provides broad protection from memory error exploits in C and C++ programs. However, previous implementations of ASR have provided a relatively coarse granularity of randomization, with many program objects sharing the same address mapping, so that the relative distance between any two objects is likely to be the same in both the original and randomized program. This leaves the randomized program vulnerable to guessing, partial pointer overwrite and information leakage attacks, as well as attacks that modify security-critical data without corrupting any pointers. To address this weakness, we presented a new approach in this paper that performs randomization at the granularity of individual program objects — so that each function, static variable, and local variable has a uniquely randomized address, and the relative distances between objects are highly unpredictable. Our approach is implemented using a source-to-source transformation that produces a self-randomizing program, which randomizes its memory layout at load-time and runtime. This randomization makes it very difficult for memory error exploits to succeed. We presented an analysis to show that our approach can provide protection against known as well as unknown types of memory error exploits. We also analyzed the success probabilities of typical attacks, and showed that they are all very small. Our experimental results establish that comprehensive address space randomization can be achieved with overheads that are comparable to coarser forms of ASR. Furthermore, the approach presented in this paper is portable, compatible with legacy code, and supports basic debugging capabilities that will likely be needed in software deployed in the field. Finally, it can be selectively applied to security-critical applications to achieve an increase in overall system security even in the absence of security updates to the underlying operating system.

Acknowledgments

We are thankful to Wei Xu for his insightful comments on the implementation issues, and the anonymous reviewers for their comments and suggestions.

This research is supported in by an ONR grant N000140110967 and NSF grants CCR-0098154 and CCR-0208877. Sekar's work was also partly supported by DARPA through an AFRL contract FA8750-04-0244.

References

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Florida, 20–24 June 1994.

[2] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference*, pages 251–262, Berkeley, CA, June 2000.

[3] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM conference on Computer and Communications Security (CCS)*, Washington, DC, October 2003.

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, Washington, DC, August 2003.

[5] CERT advisories. Published on World-Wide Web at URL http://www.cert.org/advisories, May 2005.

[6] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[7] T. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *IEEE International Conference on Distributed Computing Systems*, Phoenix, Arizona, April 2001.

[8] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic protection from `printf` format string vulnerabilities. In *USENIX Security Symposium*, Washington, DC, 2001.

[9] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2003.

[10] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.

[11] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212. Springer Verlag, June 2001.

[12] D. C. DuVarney, V. Venkatakrishnan, and S. Bhatkar. SELF: a transparent security extension for ELF binaries. In *New Security Paradigms*

Workshop (NSPW), Ascona, Switzerland, August 2003.

[13] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL http://www.trl.ibm.com/projects/security/ssp/main.html, June 2000.

[14] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems*, pages 67–72, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[15] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[16] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *USENIX Winter Conference*, pages 125–138, Berkeley, CA, USA, January 1992.

[17] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: a safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[18] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*. Linkoping University Electronic Press, 1997.

[19] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: An interpreter-based programming environment for the C language. In *USENIX Summer Conference*, pages 161–171, San Francisco, CA, June 1988.

[20] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM conference on Computer and Communications Security (CCS)*, pages 272–280, Washington, DC, October 2003.

[21] S. C. Kendall. Bcc: run–time checking for C programs. In *USENIX Summer Conference*, El. Cerrito, CA, 1983.

[22] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2001.

[23] S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Conference on Compiler Construction*, 2002.

[24] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, Portland, OR, January 2002.

[25] PaX. Published on World-Wide Web at URL http://pax.grsecurity.net, 2001.

[26] M. Prasad and T. cker Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *USENIX Annual Technical Conference*, San Antonio, TX, June 2003.

[27] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, Vancouver, British Columbia, Canada, 2000.

[28] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium*, pages 159–169, San Diego, CA, February 2004.

[29] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM conference on Computer and Communications Security (CCS)*, pages 298 – 307, Washington, DC, October 2004.

[30] J. L. Steffen. Adding run-time checking to the portable C compiler. *Software-Practice and Experience*, 22:305–316, April 1992.

[31] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego, CA, 2000.

[32] Webstone, the benchmark for web servers. http://www.mindcraft.com/webstone.

[33] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Symposium on Reliable and Distributed Systems (SRDS)*, Florence, Italy, October 2003.

[34] W. Xu, D. C. Duvarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Newport Beach, CA, November 2004.

[35] L. Xun. A linux executable editing library. Masters Thesis, 1999. available at http://www.geocities.com/fasterlu/leel.htm.