Known/Chosen key attacks against software Instruction Set Randomization

Yoav Weiss
Discretix Technologies Ltd.
POB 8718
Kefar Netter, Israel
YoavW@discretix.com

Elena Gabriela Barrantes
Escuela de Ciencias de la Computación e Informática
Universidad de Costa Rica
San José, Costa Rica
gbarrantes@ecci.ucr.ac.cr

Abstract

Instruction Set Randomization (ISR) has been proposed as a form of defense against binary code injection into an executing program. One proof-of-concept implementation is Randomized Instruction Set Emulator (RISE), based on the open-source Valgrind IA-32 to IA-32 binary translator. Although RISE is effective against attacks that are not RISE-aware, it is vulnerable to pure data and hybrid data-code attacks that target its data, as well to some classes of brute-force guessing. In order to enable the design of a production version, we describe implementation-specific and generic vulnerabilities that can be used to overcome RISE in its current form. We present and discuss attacks and solutions in three categories: known-key attacks *that rely on the key being leaked and then used to pre-scramble the attacking code;* chosen-key attacks *that use implementation weaknesses to allow the attacker to define its own key,or otherwise affect key generation; and* key-guessing ("brute-force") attacks*, about which we explore the design of minimalistic loaders which can be used to minimize the number of mask bytes required for a successful key-guessing attack. All the described attacks were tested in real-world scenarios.*

1. Introduction

Instruction Set Randomization (ISR) is a security approach that scrambles the instruction set, making it harder to inject meaningful code into a process [1, 10]. This approach imitates the protection afforded by the genetic diversity found in nature. Although theoretically sound against its threat model, any software ISR implementation (via emulation), will suffer from vulnerabilities related to the lack of hardware support, and the complexity of the emulation process. To test the extent of these vulnerabilities, we will use Randomized Instruction Set Emulation (RISE) [1] a proof-of-concept, user-side, open-source implementation of ISR.

The main contributions of this paper are:

- Identifying pitfalls which should be avoided when designing an ISR implementation.
- Describing inherent weaknesses of the ISR approach.
- Showing a way to calculate an upper limit of mask strength and demonstrating it for the IA-32 architecture.
- Suggesting a hardware-assisted solution which would be immune to the vulnerabilities above.

The paper is organized as follows: We start by describing the ISR rationale and RISE in particular in Section 2. We then proceed to describe and suggest solutions to the three attack mechanisms that were identified: Section 3 describes attacks that rely on the ability to steal/derive enough of the key-stream to inject properly-scrambled code into the process, Section 4 focuses on attacks that target the diversity mechanism itself and disable it before executing the main attack, and Section 5 explores minimalistic loaders which can be used to minimize the number of mask bytes required for a successful attack. In Section 6 we review related work, and we summarize our findings in Section 7.

2. Background

There are two areas which affect the study of software ISR vulnerabilities: the implementation of the ISR itself, and the exploit types. We start by reviewing the implementation of RISE in Section 2.1, and present vulnerability classes in Section 2.2.

2.1. RISE

RISE [1] is an ISR implementation developed using the open-source, IA-32 to IA-32 Valgrind [16] emulator. It is a pure-software implementation, running entirely in user mode. Because of efficiency considerations, the RISE and the protected process both reside in the same memory space, which is a vulnerability, given that a RISE-aware attack acting in the same memory space can subvert emulator structures.

RISE operates in two steps: (a) After the protected process has been loaded into memory, all executable portions of the process (including shared libraries) are scrambled by xor-ing each byte of the process' code with a mask. The mask could be created ahead of the scrambling, or on demand; and (b) During emulator-mediated fetch the original instructions are recovered by xor-ing the corresponding mask against the scrambled instruction.

The protective effect of RISE rests on the fact that injected code is expressed in cleartext, and therefore is scrambled during fetch. At a higher level, the rationale is that the process is running in its own "language", and foreign (injected) code is not, making the execution of the attacking instructions impossible.

RISE has two modes of operation. One is a tiled mode, in which a random mask of two or more pages is generated before execution begins, and every byte of code is scrambled (xor-ed) with a byte in the mask table. The other mode uses a one-time pad (OTP), by generating a unique mask for every code page, which results in allocation of mask space the size of the code. In this mode, the mask is generated lazily per-page, upon first access. In both modes, the security of the solution is based on the secrecy of the mask, which is in effect the cryptographic key, and therefore a possible way for a RISE-aware attack to succeed is by either leaking or guessing the key. We review existing exploit techniques that could be used to achieve this goal next.

2.2. Exploit types

There are two issues to consider when examining RISE-aware attacks: the first is whether there is some way of undermining the structure of RISE to execute code, and the second is how likely are we to find a vulnerability in the wild that enables the attack. One way of assessing this likelihood would be to show that there is a way to construct the attack for most classes of low-level vulnerabilities. Therefore, we briefly review the possible vulnerabilities here.

The first observation is that for leaking the mask it is enough to find a pointer to a structure that will eventually be sent back to the user (e.g. an error or data-displaying string), and redirect this pointer to either the mask or the code area. These pointers can be found in the stack, and other data areas of the process. If the process sends data back to the user at any time, there is at least one such pointer somewhere. The second observation is that a hybrid data-code attack could work if the data structures of RISE could be subverted prior to executing code. In both cases, an attack to change a pointer does not require the execution of any code, so RISE will not interfere. Chen et al. [5] presented evidence of the feasibility of data-only attacks and the ubiquity of vulnerabilities that allow them. Karger and Schell [9] showed several real alternatives. If the attacker has access to a format string vulnerability, data can be leaked from an arbitrary location by using the correct combination of conversion specifiers [15]. It is also possible in some cases to turn a stack vulnerability in which no useful pointers could be found into a format string vulnerability by using Durden's technique [6].

In sections 3, and 4, we will show with specific attacks how these exploits can be used to bypass RISE.

3. Known-Key Attacks

The first type of attack against RISE we will explore is the stealing of the mask, as it is the more natural approach. There are two sources for this information: the mask itself, and the scrambled code. In Section 3.1 we explore the possible ways for obtaining the mask from the structures that store it, and in Section 3.2 the extraction of the key from the scrambled code is examined.

3.1. Direct key extraction

Although the mask is not hidden deeply in the process, a complicating detail is that to "steal" the key for the memory area where the injected code will run, this key has to exist, and in RISE this is not always the case. In one time pad (OTP) mode, RISE generates the key lazily: if the memory area is not part of pre-existing code areas, the key for that area simply does not exit. In tiled mode, the key pre-exists, but there is a small uncertainty about which tile would be used for a particular memory area. However, even in OTP mode the laziness works on page-sized chunks, so parts can still be stolen. We will study the stealing of a tiled mask first, and then consider the OTP case.

In tiled-mode, the entire mask is read from `/dev/urandom` before the executable is mapped to memory, so unless an additional memory randomization is used, the mask appears right before the executable in memory, in a fixed location. Several methods could be used to steal the mask and use it to properly scramble the attack code. We will show two examples: one using a format-string vulnerability, and another one using a stack overflow.

3.1.1 Tiled-Mode key stealing using format strings

Our first example uses a format-string attack. Consider the following trivial format-string vulnerability:

```
main(){
   char x[1000];
   while (fgets(x,1000,stdin))
      printf(x);
}
```

A typical code-injection exploit would change the address of `fgets()` or `printf()` to point to an address somewhere in the middle of `x[ ]`, append a sled of NOPs followed by a shellcode, and wait for the string function to be called, which will execute the injected code. Such an exploit will not work with RISE because the shellcode will be scrambled upon fetch, and the CPU will end up executing random instructions, likely to result in a crash.

A RISE-aware attacker could use the following strategy instead: find the mask corresponding to the exploit location inside x[ ], scramble the exploit code using the stolen mask, and then use the traditional exploit. To find the mask, it just needs to find address of the relevant mask, and display it (with %s). S_A, the address of the specific piece of mask for the shellcode is given by Equation 1, where P_A is the page address of S_A, T_N is the tile number used for the shellcode address, Psz is the page size, and S_O is the shellcode offset.

$$S_A = P_A + T_N \times Psz + S_O \quad (1)$$

The only unknown in Equation 1 is T_N, which is a number between zero and the maximum number of mask pages. For a mask-size of 8192 (two pages), T_N can be 0 or 1, so it can be guessed with 0.5 probability of success. Note that a more sophisticated exploit could avoid any guesswork by dumping the table that associates mask pages to code pages, and checking which mask page corresponds to the page where the shellcode is stored.

The mask string returned to the attacker must be as long as the shellcode. However, since strings returned via format strings will stop the data dump when encountering a null character, a zero in the mask may cause the returned value to be too short for the needs of the attacker. This limitation can be circumvented in several ways:

1. Iteratively retrieving zero-enclosed chunks of the mask: If the attacker gets a string of N bytes, byte $N + 1$ must be zero. The data dumping can be repeated using the address of $N + 2$, and so on, until enough bytes are retrieved.
2. Extracting a different part of the mask, in an attempt to find a chunk large enough without zeros. The shellcode has to be moved to align it up with the recovered mask.
3. Using a two-phased shellcode, where the first part is small enough to be used with the extracted mask, and then using it to load and scramble the second (larger) part of the shellcode.
4. Trying again on another instance of the server (assuming it forks), until finding a mask large enough without zeros.

Appendix A in [20] shows the python script we used to test a successful exploit against RISE based on this method.

3.1.2 Tiled-Mode key stealing using stack overflow

Although format string vulnerabilities offer the most convenient way to steal the key, the attack can also be executed using a stack or heap overflow vulnerability, and can be generalized to other pointer corruption vulnerabilities. As in the previous section, we will use a toy stack vulnerability as reference:

```
void f() {
char *p;
char buf[100];
   p = buf; gets(buf);
   while (*p) putchar(*p++);
   putchar('\n');
   fflush(stdout);
}
main() {
   while(!feof(stdin)) f();
}
```

The most common code-injection exploit would send a shellcode, pad all the way to the return address of `f( )`, and overwrite it with the address of buf[ ]. As before, this will not work with RISE because the shellcode would be scrambled upon fetch, and a random sequence of instructions would be executed. A RISE-aware attacker can use `(*p)`, which is stored in the stack and points to a string that will eventually be sent back to the user. Instead of overwriting the return address, the strategy would be to overwrite `*p` with the address of the mask we are interested in, and then wait for the function to send the mask data back. This example is oversimplified but any pointer (even on a different frame) up the stack, would work. Appendix B in [20] contains the exploit we used to successfully defeat RISE using this strategy.

The main problem with this approach is that there is no guarantee that the stack will contain a useful pointer in all cases, or the attacker might not want to damage other frames in the stack (which is unavoidable during the overflow that overwrites the pointer). There is an alternative way of exploiting a stack vulnerability in these cases. We use the following code to exemplify the point:

```
int f(char *buf1, int len) {
char buf2[100];
int c = read(0,buf2,200);
   if (c == -1) return -1;
   if (buf1) return memcmp(buf1,buf2,len);
   else return f(buf2,c);
}
main() {
   printf("Comparing input pair:\n");
   exit(f(0,0) != 0);
}
```

This program never sends any non-constant data to the user, and never sends anything to the user after the overrun occurs. Therefore, it is not vulnerable to the previous attack. However, the attacker can still use Durden's technique [6], which is explained using the example given. The `printf()` (in `main()`) is close enough to the buffer overrun (in `f()`), so overwriting the lowest byte of the return address with the offset of the call to `printf()` can make the program send mask data back to the attacker, and fall-through to the `f(0,0)` call, allowing the attacker to overflow the stack again and utilize the stolen mask.

Here is a snippet of assembler code from `main()` of the above program:

```
# printf("Comparing input pair:\n");
8048498:  movl $0x8048604,(%esp,1)
804849f:  call 804830c <_init+0x48>

# exit(f(0,0) != 0);
80484a4:  movl $0x0,0x4(%esp,1)

# ...
80484b3:  call 8048414 <f>
80484b8:  test %eax,%eax
```

Note that the return address of `f(0,0)` is `0x080484b8`, which happens to be in the same page of the call to `printf()`, in `0x0804849f`. An attacker can exploit this as follows:

1. Send a first input which does not overflow the stack but is a format string (this will be used later to extract mask bytes). For example, `"\x61\x46\x13\x40%8s"` (extract a string from `0x40134661`, where some mask bytes reside).

2. Send a second input which overflows the stack, overwriting the low byte of the return address with `0x9f` (the offset where printf is called): `"\x9f"*125`.

3. At this point, the program leaks mask bytes and falls back into `f( )`, so instead of calling `exit( )`, it reads a third input. Send as third input the shellcode prescrambled with the stolen mask.

Mask-leaking techniques work against heap overruns [8] as well, by exploiting the `unlink()` macro in `dlmalloc()` to overwrite a pointer of some data sent to the user. An additional attack method can be based on cache attacks [14], but we intentionally left it out as the original RISE threat model does not include local attackers.

3.1.3 Testing Tiled-Mode key stealing

The feasibility of leak attacks was verified using the synthetic vulnerabilities shown, the classical *Site Exec Command* format string vulnerability [4], and two current proprietary applications, one for a stack overflow, and one for a heap overrun vulnerability. As part of the test, two such exploits were modified to work against a RISE-protected system, using methods similar to the ones discussed in previous sections the above. We conclude that real-life exploits can often be improved to work against RISE.

3.1.4 Stealing the key in OTP mode

The attacks discussed above are useful against RISE running in tiled mode. Variants of these attacks can also be used against RISE in OTP mode, but with a lower probability of success. In OTP mode, the mask is generated lazily, but always for an entire page. This means that an attack must be done in two stages. First, we need to cause RISE to generate a mask for a writable page, then, after stealing the mask, we need to inject the (masked) shellcode into the target page.

One possible strategy to trigger mask generation without crashing the process is to cause a single instruction to execute in the target page, and then returning, without causing a crash or otherwise damaging the process. A clear candidate is the RET instruction (`0xc3` for the IA-32 processor family) which is one byte in length, and exits back to the application. The process can be altered to call the beginning of the desired (data) page by altering some function pointer (such as a GOT entry) or by overwriting a return address on the stack and the word above it on the stack to point to an appropriate location in the code. After being redirected, RISE will start to execute whatever random instructions are created by xor-ing the newly created mask bytes with the data on that page. To succeed, the attack needs to hit a sequence of zero or more non-destructive random instructions, followed by a randomly generated RET.

The worst case for the defender is that RET executes as the first random instruction of that page. The probability of this event is $\frac{1}{256}$, which will require to repeat the attack about 256 times, well inside the acceptable range for brute-forcing without raising alarms in the target system. However, the overall probability of hitting a RET after some non-destructive instructions is even higher, as attested by the Markov-Chain analysis in [2]. After success, a new

mask that is not covering real code is created, and the target process continues running, which allows the attacker to steal the mask for this (data) page using the techniques described in section 3.1.2 for tiled mode.

3.1.5 Portability of key-stealing attacks and possible solutions

The key stealing attacks described are ISR-implementation specific. It is likely that they would be feasible against any user-mode pure-software implementation of ISR, where the mask has to be kept in the same memory space used by the program. Hardware implementations are likely to be immune against direct key-stealing because there is no need for the key to be accessible to the process. For software implementations of ISR, some solutions are offered:

1. Make the key unreadable during the execution of non-RISE code. We were able to easily harden RISE against this specific attack (direct key-stealing) by changing the access rights of the memory addresses containing the mask to PROT_NONE, making it readable only when it is actually needed. This solution slows down the program during the warm-up period but once most of the used code is in the (plaintext) cache, performance is no longer affected. Note that this fix is not thread-safe and can only protect a single-threaded server, but could be solved by appropriate locking at an additional performance cost during warm-up.
2. Randomize the location of the key. This solution, mentioned in [2], in reality requires the randomization of the position of any component that could indirectly lead to the mask address. That includes the table that contains the pointers to the masks, and anything that points to it from stack or heap, so it requires full address space layout randomization (ASLR) rather than just randomizing the key location. Otherwise, an attacker could steal the non-randomized component and traverse from there until mask is found. Bhatkar et al. [3] have recently published a tool that thoroughly randomizes the internal layout of a program by recompiling it. It could be possible to randomize just the layout of the emulator, which would possibly make these key-stealing attacks less likely.
3. Separate the memory space between emulator and emulated processes. However, this option is extremely expensive and is seldom used for emulation.

Even if stealing the key directly is made very difficult, there is an alternative method of obtaining the key, which is through the scrambled code itself. This approach will be discussed in the next section.

3.2. Known-Plaintext key extraction

A software ISR has to use a stream-cipher in order to allow efficient byte-level access to the scrambled memory space. RISE uses XOR with a mask originating from `/dev/urandom`. As with any such stream-cipher, the ISR is therefore vulnerable to known-plaintext attacks if keys are reused. In the case of RISE, tiled-mode reuses keys, so this attack becomes feasible.

An attacker with knowledge of the binary executable (or one of the shared objects used, such as `libc.so`) can dump a piece of code encrypted by the global key, using one of the methods described earlier, xor it with the known plaintext, and get the key. In particular, any user in possession of a binary (be it open-source or not) can examine the machine code of the program, and use it to launch this attack against a system using the same program.

System-level ASLR solutions such as PaX [13] cannot make this attack any less reliable since they randomize areas such as stack, heap and `mmap` base, but leave the main process executable in its original location. The attacker can still dump pieces of the scrambled executable without any guess-work. Even in the unlikely event that the executable is unknown, the attacker can reliably walk through it, find a known structure, such as the Global Offset Table (GOT) [19], and get addresses of library functions, canceling the effect of `mmap` randomization and gaining access to xor-encoded `libc` code -an almost universal known-plaintext.

Existing implementations that randomize addresses in the executable itself are seldom used because they require relinking the application. Examples of such implementations are PaX [13] with ET_DYN-linked binaries and Exec-Shield [12] with PIE position-independent executables. Currently, there is no reliable way to fully randomize addresses without relinking, and most Linux distributions are not ET_DYN-ed.

A full ASLR (with relinking) makes this attack somewhat harder, but not infeasible, because

$$XOR(XOR(a,m),XOR(b,m)) = XOR(a,b),$$

so even if the mask (m) is not known, we know a and b (pieces of the code). Unlike ret-to-libc exploits, this attack does not require an exact location in the code. Any known-plaintext encrypted code is usable. The steps for executing such an attack are the following:

1. Gather all known-plaintext code (libraries and the executable if known).
2. For each page of code P, make

$$X=XOR(P, P+<\text{page align(mask-size)}>)$$

3. Create a dictionary D with X as keys and P as corresponding values.
4. Dump a random page R (iteratively if required), and page R' which is <page align(mask size)> bytes above R.
5. If D(XOR(R,R')) exists, then

   key =XOR(start from R, start from P)

6. If not, goto 4

The probability of success is derived from the density of code (the percentage of pages populated by code) in memory space. As found in [2], the code density of most processes is below 5%. At a 5% density, each iteration has $\approx \frac{1}{20}$ probability of finding the key. An iteration that fails might crash the process if it hits unmapped memory, but will fail silently if it hits heap, stack, bss, data, or any other non-code item. Therefore, the probability of successfully acquiring the key using this algorithm is bound to be better than $\frac{1}{20}$ in this case, which is rather feasible in real world situations.

If the vulnerability is a format-string one, or a stack overrun that can be converted to a format string attack (as described in section 3.1.2), this algorithm can be improved significantly. The attacker can dump a piece of the stack to find return-addresses on the stack, which point directly to code. Feeding such addresses into the algorithm above, instead of using a random R perform close to 100% reliability. Other derandomization attacks against full ASLR may be used to improve the reliability of this method [17].

3.2.1 Portability of known-plaintext attacks and possible solutions

Unlike the key-stealing attack, the known plaintext attack does not rely on the specific implementation of RISE. Any software ISR implementation that reuses keys can be attacked this way with high probability of success. Furthermore, if a hardware implementation uses a simple stream cipher such as XOR with pseudo-random keys, and reuses key bytes, then it will not be immune to this attack.

We propose the following solutions to direct key stealing attacks:

1. Make the original code unreadable after scrambling. Allow reads only during each fetch operation. Exceptions for self-references (unavoidable in emulation) must be allowed. A problem with this approach is that read, write, and execute rights are granted at a page granularity, but the ISR has to work at a byte granularity (at least on IA-32). Mixed code/data pages have to be left readable. Therefore, this solution will be effective only if full ASLR is in effect, so mixed pages are not easily locatable. An additional problem of this solution is its efficiency, as the extra calls to `mprotect` will slow down the warm-up phase even further. This approach is also difficult to port to hardware, as an efficient hardware ISR has to work at fetch-level and can hardly be checking per-page state for each fetch. Finally, some CPUs, including IA-32 do not support execute-only pages.

2. Avoid tiled-mode and use OTP instead. If the key is not reused, plaintext knowledge is useless. However, OTP requires O(code size) RAM, which might be prohibitive in some situations.

3. Increase the entropy of the tiles. The key to this approach is to find a non-invertible function that allows the ISR to "personalize" the tiles. This function can be expensive in space (if we add additional per-page secrets, for example), or expensive in time (if the function has to make complex calculations for each fetched byte). Some of these functions have implicit vulnerabilities that require the use of full ASLR on top. However, it is possible to find functions that do not suffer from this problem. We present one such function below.

   In a hardware implementation of RISE, the mask-table can be dropped completely and replaced with f( ), a function of the instruction pointer (EIP in IA-32) and a stored secret (S): f(EIP,S), where f() is a hash or similar function that must be very secure. f() is defined to be secure if even if the attacker retrieves a large number of f(EIP_i,S) this will not make it easier to calculate f() for unknown values of EIP, or to derive the stored secret. LRW encryption mode [11] may be a suitable alternative. Implementing this solution in software would be prohibitively slow, but if a hardware accelerator can calculate the secure function at clock speed, the system would be immune to key-deriving attacks in most cases. However, programs that dynamically load and unload shared objects are an exception, because they may use a page for a writable object after unloading scrambled code from it, leading to reuse of key. Such programs must take care to avoid mapping a page as writable if it previously contained code. This can be done using `dlopen( )` and `dlclose( )`, and keeping freed pages mapped for their own future use rather than unmapping them.

Note that while solutions 2 and 3 do not require a full ASLR (with relinking), it is still advisable to use it. Any system that relies on RISE without ASLR (or with a partial ASLR) is likely to be vulnerable to ret-to-libc attacks, and as the next attack will demonstrate, ret-to-libc may even be escalated to code-injection.

4. Attacking the immune system: Chosen-Key attacks

An attacker could take a different approach by attacking the diversity mechanism itself. Such attacks, by nature, are implementation-specific, but can be highly effective. In the current implementation of RISE, for example, it is surprisingly simple to disable the protection in OTP mode. The mask is generated lazily by reading from `/dev/urandom`, which is kept open for the life of the process. In the test-case used, it was always associated with file-descriptor 4. Given that RISE starts its life as the first shared library to be loaded for the protected process, it is highly likely that this descriptor would always be the same. Note that file-descriptors are shared between RISE and the protected process because of the memory-space sharing. If the process calls `dup2(0,4)`, `/dev/urandom` is replaced with stdin, allowing the attacker to choose the key for masking the pages (hence the "chosen-key" label). In particular, if the attacker uses zeros, it can run any shellcode unchanged as RISE has been effectively turned off. Appendix C in [20] shows an implentation of this attack.

Although the attack described is just an example, there is a richness of structures in RISE (and in any software ISR, at that) that could potentially be subverted to allow the use of whatever key the attacker wishes to use. There are two interesting uses for this class of attacks:

1. Hybrid ret-to-libc plus injection. On RISC platforms, ret-to-libc attacks are limited by the fact that registers often do not point to the required strings in memory at the time of attack, and the attacker seldom has a way to control their content. However, having the values 0 and 4 in the registers is much more likely than having a register point to the string `"/bin/sh"`. In such situation, disabling RISE by a `ret-to-dup2()` will enable injection of shellcode that does not require parameters, thus fully compromising the process.

2. Create a tiny loader when we are only able to acquire or guess a small number of mask bytes. Incremental-guessing methods such as the ones described in [18] can guess mask bytes but at a relatively-high cost per-byte. Sovarel et al. used a 100-byte loader that was able to execute code of any size, although they required that the ISR did not regenerate the mask after each attack (which RISE does). We show that a loader that disables RISE using `dup2()` and executes any code, can be as small as 13 bytes and run any attack code, unchanged.

   This attack demonstrates that the cryptographic strength of the current implementation of RISE is limited to 13 bytes, despite using a mask of sizeof(code). It also demonstrates that the current implementation is vulnerable to hybrid attacks rather than just pure ret-to-libc ones.

The diversifier (the component responsible for introducing diversity in RISE) can be disabled by other methods as well, such as overwriting the mask-table pointer, replacing the mask with known data. This class of attacks would work against tiled-mode as well.

4.1. Portability of chosen-key attacks and possible solutions

Chosen-key attacks are implementation-specific by nature. The `dup2()` example probably will not work against other implementations, but then different software ISR implementations will have other vulnerabilities in this class. These vulnerabilities demonstrate an important point: future RISE and other ISR implementations must be carefully audited for ways in which the code could affect the diversifier behavior. An attack against the diversifier will compromise the entire system.

Once chosen-key attacks are identified, it is usually trivial to mitigate them. For example, RISE could `fstat( )` to check whether fd 4 is still `/dev/urandom` before reading from it, and reopen it if it has been replaced. Mutex may be required to make it thread-safe. Other diversifier-disabling attacks can be mitigated in analog ways. The important point for future implementations is that the protected code should never be able to affect the behavior of the diversifier.

If mask capture and data structure subversion are no longer available, it is always necessary to consider a third possibility: pure guessing (brute-force) attacks. We examine this class of attacks in the next section.

5. Key-Guessing Attacks through minimal payloads

This section describes three types of tiny loaders which can be used as payload for key-guessing or key-leaking attacks. The purpose of this section is to show the minimum number of mask bytes that need to be uncovered for successfully mounting an attack. The presented loaders are specific to IA-32, but can be written for RISC processors at a higher price in mask bytes. All three methods use 2-bytes execution buffers. As it turns out, any useful shellcode can be expressed without using instructions longer than 2 bytes, by converting the code to a stack-machine.

5.1. The 7-byte loader

The first loader requires leaking or guessing 7 mask bytes. The first two bytes in the loader are self-modifying and used

for loading code (scrambled by the first two mask bytes), two bytes at a time, from the stack. The other 5 bytes are the code for loading additional code from the stack and "returning" into it. It uses a single page so it can work against both tiled and OTP implementations. However, it does not work against the Valgrind-based RISE because Valgrind breaks self-modifying code in the instrumentation process.

The 7-byte loader will work on any ISR implementation that allows modifying code between executions. It has been verified to work against a synthetic implementation that restricts execution to a buffer of 7 bytes but allows self-modifying code. The code for the loader is shown below:

```
5d        pop %ebp
90        nop
5e        pop %esi
89 75 01  mov %esi,0x1(%ebp)
c3        ret
```

The loader expects the stack to start with an address 1 byte before loader. It loads into itself, and executes 2-byte sequences. The (reversed) stack construction is described below:

```
# To "return" to loader:
&loader
# To set ebp:
&loader-1
# Load 1st instruction and restore
loader[2:3]:
1st_instruction(2-bytes)+loader[2:3]
# To "return" to the new instruction:
&loader
# Argument (only if 1st inst.  is a pop):
[arg]
...
# Load next instruction and restore
loader[2:3]:
Nth_instruction(2-bytes)+ldr[2:3]
# To "return" to the new instruction:
&loader
# Arg.  (only if Nth inst.  is a pop):
[arg]
```

This loader requires a stack overflow, knowledge of at least 7 mask bytes, and a system that supports self-modifying code. It has a success probability of $\frac{1}{\text{number of mask pages}}$. For example, with a mask of 8192 bytes, it works with 0.5 probability.

5.2. The 4-byte loader

This variant uses the same technique as the 7-byte loader, and works with a slightly lower probability, but only requires leaking a single 4-byte integer from the mask. The loader is split into two 4-byte chunks, to be placed in two consecutive pages. Each chunk uses RET to return to the other chunk:
First chunk:

```
5d        pop %ebp
90        nop
5e        pop %esi
c3        ret
```

Second chunk:

```
89 75 01 mov %esi,0x1(%ebp)
c3        ret
```

The chunks get executed in the same offset of two different pages, so they should be 4K apart. As before, the loader expects the stack to start with an address 1 byte before loader. It loads into itself, and executes 2-byte instructions. The (reversed) stack should be constructed as follows:

```
# to "return" to loader:
&chunk1
# To set ebp:
&chunk1-1
# Load 1st inst.  & restore chunk1[2:3]
1st_instruction(2-bytes)+chunk1[2:3]
# To "return" to the second phase:
&chunk2
# To "return" to the new instruction
&chunk1
# Arg.  (only if 1st inst.  is a pop):
[arg]
...
# Load next inst.  & restore chunk1[2:3]
Nth_instruction(2-bytes)+chunk1[2:3]
# To "return" to the second phase:
&chunk2
# To return to the new instruction:
&chunk1
# Arg.  (only if Nth inst.  is a pop):
[arg]
```

This loader requires a stack overflow, knowledge of at least 4 mask bytes, and a system that supports self-modifying code. Unlike the 7-bytes loader, it assumes reuse of mask in two different pages, so it is only effective against tiled-mode. Its success probability is $\left(\frac{1}{\text{number of mask pages}}\right)^2$ because it needs both pages to share a mask. For example, with a mask of 8192 bytes, it works with 0.25 probability.

5.3. The 3-byte loader

The 3-byte loader does not require self-modifying code, and therefore works against RISE. It requires an unbound (or

very large) stack overflow and works at a lower probability of success than the previous loaders, but still of a low polynomial order. It uses 2-bytes execution buffers in consecutive pages, and places a RET after each buffer (hence the 3 bytes). The stack is responsible to direct the RETs to the next page after each execution. The loader requires at least $\frac{\text{sizeof(shellcode)}}{2}$ pages. It sometimes requires more pages because it skips pages for which the address contains a zero (due to string-functions limitation when overwriting the stack).

Because the probability of success of this loader is polynomial on the number of instructions, we optimized the shellcode used in the other two exploits, and reduced its size to 12 sequences of 2 bytes each.

This loader requires a large (or unbound) stack overflow, knowledge of at least 3 mask bytes. It does not require self-modifying code. It assumes that several pages share a mask, so it only works against tiled-mode.

The probability of success for this attack is $\left(\frac{1}{\text{number of mask pages}}\right)^{\text{number of instructions}}$. A typical `execve()` shellcode uses 12 instructions, so the success probability is $\left(\frac{1}{\text{number of mask pages}}\right)^{12}$. For example, with a mask of 8192 bytes, the probability of success is $\frac{1}{4096}$, which is still feasible in real-world attacks. Empirical tests against RISE show slightly better results: $\frac{1}{4006}$.

Appendixes D, E, and F in [20] present sample exploits using the 7, 4, and 3-byte loaders respectively.

5.4. Analysis of key-guessing attacks using loaders

With tiny loaders such as those described above, it becomes feasible to launch brute-force key-guessing attacks against ISR, even in situations where leaks (as described earlier in this paper) are not available. We can set an upper limit on the strength a software ISR can hope to achieve, and show that increasing mask strength beyond a certain point would have no effect on security.

In OTP mode, the 3-byte and 4-byte loaders will not work, so a key-guesser will have to use the 7-byte loader and launch an average of 2^{7*8} attacks, guessing a 56 bits key. Crashing an application 2^{56} times will probably take too long and attract too much attention in most real-world situations, so it can be considered reasonably secure. Unfortunately, OTP mode doubles the memory consumption of the code and is not likely to be used in real-world situations.

In tiled-mode on a system that does not allow self-modifying code, only the 3-byte loader can be used, and it requires 12 pages for running a useful shellcode. Its probability of success when key-guessing is $2^{3*8} * (\text{number of mask pages})^{12}$. With a minimal mask size of 8192, it means 2^{24+12}, so an average of 2^{36} guess-attempts are required. This is much less than the 56 bits of OTP mode, and may be feasible in some real-world situations.

In tiled-mode on a system that allows self-modifying code, the 4-bytes loader can be used, and will often perform better than the 3-bytes loader because it only needs 2 pages. The probability of success when key-guessing is $2^{4*8} * (\text{number of mask pages})^2$. With a mask size of 8192, it means 2^{32+2}, so an average of 2^{34} guess-attempts are required. This is probably feasible in real-world attacks when attacking an unattended server over a weekend, or an embedded device that does not send crash reports to a system administrator.

While discussing these attacks, it has been suggested that the security of a software ISR may be improved if the mask is not uniformly-random and makes it harder to inject certain instructions. For example, if 0xC3 is made more common, it will be harder to include a RET in the code because a scrambled RET is more likely to be an uninjectable zero. However, non-uniform randomization will be easy to bypass and cause more harm than good (for a discussion on this point, refer to Appendix G, in [20]).

Given these upper limits (34 or 36 bits in tiled-mode, 56 bits in OTP mode), one could argue that RISE is somewhat limited when protecting IA-32. Note, however, that on a RISC architecture these numbers will be somewhat larger.

Tiled-mode should probably be avoided where security matters, since it is vulnerable to relatively fast key-guessing, as well as leak attacks as demonstrated in the first part of the paper. OTP mode is a bit more immune to key-guessing as well as to leaking attacks, but requires too much memory to be feasible.

Therefore, a third mode is required, as suggested earlier in this paper. This mode will use a secure address-sensitive encryption function, such as LRW or some block cipher in CTR-mode, for calculating the mask for each instruction.

6. Related Work

Diversifying, or randomizing approaches to improve system security have been discussed for a long time [7]. Although the original intention of these defenses was to provide a barrier against the rapid expansion of exploits over identical systems, there seems to be interest in using them as first-order defenses, so in order to test their reliability the analysis of several attacks against such systems have been published.

In 2004, Shacham et al. [17] presented a brute-force guessing attack against the PaX ASLR [13]. ASLR randomizes the location of the libraries and some of the data segments of a process. The success of the attack relied on the fact that only 24 bits were using for the randomization, and that PaX was not re-randomizing the position of the

segments after each crash, so the attack could learn from its failures.

Sovarel et al [18] introduced an attack against RISE that successfully guessed 100 mask bytes in 8636 attempts and implemented a 100 bytes virtual machine to execute a payload of an arbitrary size. Unfortunately their attack required that the key stay the same across crashes, which has never been true for RISE, which regenerates the masks after each crash. Therefore, their attack was tested against an artificially crippled version of RISE that did not re-generate the keys. This paper not only presents guessing attacks that do not depend on the permanence of the key, but also perpendicular attacks that rely on data capture, instead of guessing to succeed.

7. Conclusions

Several methods of stealing mask bytes and injecting code into a RISE-protected process were presented in this paper. The paper explains why these methods would often be feasible in real-life situations. An attack against the diversifier itself was also demonstrated, effectively disabling RISE. Minimal loaders were developed to derive an upper limit on the strength of any ISR implementation that has to honor the byte-per-byte interpretation required on a Cisc architecture such as IA-32.We showed that on any such system, the ISR would never be stronger than 56 bits. Solutions to the demonstrated weaknesses were proposed and evaluated.

ISR is a promising approach to application security, and should be further researched to uncover and fix vulnerabilities. Any ISR implementation has to be carefully examined because, as this paper shows, implementation details can make or break the security of the system.

Acknowledgements

The authors would like to thank Hagai Bar-El and Rony Shapiro for the helpful discussions, comments and reviews that made this paper possible. In addition, the authors gratefully acknowledge the partial support of the Santa Fe Institute.

References

[1] E. G. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*, pages 281–289, New York, NY, USA, October 27-31 2003. ACM Press.

[2] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanovic. Randomized instruction set emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):3–40, February 2005.

[3] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, August 2005.

[4] CERT Coordination Center. Cert advisory ca-2001-33. multiple vulnerabilities in WUFTPD, 2001.

[5] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the Usenix Security Symposium*, page 177192, 2005.

[6] T. Durden. Bypassing PaX ASLR protection. *Phrack*, 59(9), June 2002.

[7] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[8] M. Kaempf. Vudo malloc tricks. *Phrack*, 57(8), 2001.

[9] P. A. Karger and R. R. Schell. Multics security evaluation: Vulnerability analysis. Technical report, HQ Electronic Systems Division, http://csrc.nist.gov/publications/history/karg74.pdf, June 1974.

[10] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 272–280, New York, NY, USA, October 27-31 2003. ACM Press.

[11] M. Liskov, R. Rivest, and D. Wagner. Tweakable block ciphers. *Lecture Notes in Computer Science*, 2442(CRYPTO'02):3146, 2002.

[12] I. Molnar. Exec Shield - new linux security feature. *NewsForge*, May 2003.

[13] PaX team. Documentation for the PaX project. In *http://pax.grsecurity.net/docs/index.html*, 2006.

[14] C. Percival. Cache missing for fun and profit. In *http://www.daemonology.net/papers/htt.pdf*, 2005.

[15] Scut and Team Teso. Exploiting format string vulnerabilities. In *http://http://www.team-teso.net/articles /formatstring/*, September 1 2001.

[16] J. Seward and N. Nethercote. Valgrind, an open-source memory debugger for x86-gnu/linux. In *http://valgrind.org/*, 2006.

[17] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM Press.

[18] A. Sovarel, D. Evans, and N. Paul. Where's the FEEB?: The effectiveness of instruction set randomization. In *Proceedings of the Usenix Security Symposium*, pages 145–160, 2005.

[19] Tool Interface Standards Committee. *Executable and Linking Format (ELF)*, May 1995.

[20] Y. Weiss and E. G. Barrantes. Known/chosen key attacks against RISE: Attacking the immune system. Technical Report TR-ECCI-01-2006, E.C.C.I., Universidad de Costa Rica, http://www.ecci.ucr.ac.cr/, September 2006.