Surgically returning to randomized lib(c)

Giampaolo Fresi Roglia† Lorenzo Martignoni‡ Roberto Paleari† Danilo Bruschi†
Dipartimento di Informatica e Comunicazione† Dipartimento di Fisica‡
Università degli Studi di Milano Università degli Studi di Udine
{roberto,gianz,bruschi}@security.dico.unimi.it lorenzo.martignoni@uniud.it

***Abstract*—To strengthen systems against code injection attacks, the write or execute only policy (W⊕X) and address space layout randomization (ASLR) are typically used in combination. The former separates data and code, while the latter randomizes the layout of a process. In this paper we present a new attack to bypass W⊕X and ASLR. The state-of-the-art attack against this combination of protections is based on brute-force, while ours is based on the leakage of sensitive information about the memory layout of the process. Using our attack an attacker can exploit the majority of programs vulnerable to stack-based buffer overflows *surgically*, i.e., in a single attempt. We have estimated that our attack is feasible on 95.6% and 61.8% executables (of medium size) for Intel x86 and x86-64 architectures, respectively. We also analyze the effectiveness of other existing protections at preventing our attack. We conclude that position independent executables (PIE) are essential to complement ASLR and to prevent our attack. However, PIE requires recompilation, it is often not adopted even when supported, and it is not available on all ASLR-capable operating systems. To overcome these limitations, we propose a new protection that is as effective as PIE, does not require recompilation, and introduces only a minimal overhead.**

I. Introduction

In 1988 the first buffer overflow vulnerability was used to compromise thousands of systems [1]. After twenty years, applications are still vulnerable to the same type of vulnerabilities, although today it is more difficult to abuse them because of the advances in the defensive technology. However, well motivated attackers still succeed in their intent.

Write or execute only memory pages (W⊕X) and address-space layout randomization (ASLR) are two strategies nowadays adopted in combination on most UNIX distributions [2], [3]. The former ensures that no memory page is writable and executable at the same time. The latter randomizes, at runtime, the address of certain components of a process (e.g., the stack, the heap, and shared objects). Although their combination is believed to provide a good protection against code injection attacks, the belief is not completely true. Researchers have demonstrated that these protections can be defeated by patient attackers [4]. The state-of-the-art approach to exploit stack-based buffer overflows on systems protected with W⊕X and ASLR involves mounting a return-to-lib(c) attack [5] repeatedly, in a brute-force fashion. Indeed, on 32-bit architectures (e.g., Intel x86) ASLR is weak because of low randomization entropy. Hence, with a relatively small number of attempts an attacker can guess the base random address at which a certain library is loaded and then successfully mount a return-to-lib(c) attack. However, a brute-force attack can easily rise alarms (e.g., because of a large number of crashes) and automatic mechanisms can be used to impede the attacker [6]. Furthermore, it is unfeasible to perform such an attack on a 64-bit architecture, because the number of address bits available for randomization is too high.

In this paper, we present a new approach to exploit stack-based buffer overflows in programs protected with both W⊕X and ASLR. Our attack is an information leakage attack that exploits information about the random base address at which a library is loaded, available directly in the address space of the process, and is not avoidable. Contrarily to the aforementioned brute-force attack, with ours an attacker can subvert the execution of a vulnerable program and perform a return-to-lib(c) with *surgical precision*, i.e., in a single shot. Furthermore, our attack works independently of the strength of randomization (i.e., it works on 32 and on 64-bit architecture), and it is applicable to any position dependent executable. The impact of our attack is not negligible, since the majority of executables found in modern UNIX distributions belong to this class.

In the second part of the paper we analyze the effectiveness of other existing protections, that complement W⊕X and ASLR, at mitigating our attack. Our conclusion is that ASLR is really effective only when used in combination with position independent executables (PIE) [7], and therefore the address space of both shared objects and executables is randomized. Unfortunately, modern UNIX distributions still do not widely adopt position independent executables. We believe the reason is to avoid performance penalties, but there are no commonly understood motivations. Since recompilation is necessary to transform a position dependent executable into a position independent one, we propose a new protection that is as effective as PIE at stopping our attack, does not require recompilation of any executable, and introduces only negligible overhead. The proposed mitigation technique can be used to protect users of operating systems with ASLR and PIE, but that still have to adopt PIE on large scale (e.g., all GNU/Linux distributions). Moreover, our protection can be used by users of operating systems with ASLR, but lacking PIE (e.g., OpenBSD), and by users of programs with no possibility of recompilation.

In summary, the paper makes the following contributions.

1) A new attack to exploit stack-based buffer overflows on systems protected by ASLR and W⊕X.
2) An analysis of the executables found in some of the most popular UNIX distributions to demonstrate the wide applicability of the attack.
3) A study of the effectiveness of existing protections, complementary to W⊕X and ASLR, at mitigating our attack.

```
void sanitize(char *str, int len) {
  char newstr[128];
  int newlen = 0;

  for (int i = 0; i < len; i++) {
    if (str[i] != ...)
      newstr[newlen++] = str[i];
  }
  ...
}
```

Fig. 1. Sample vulnerable program

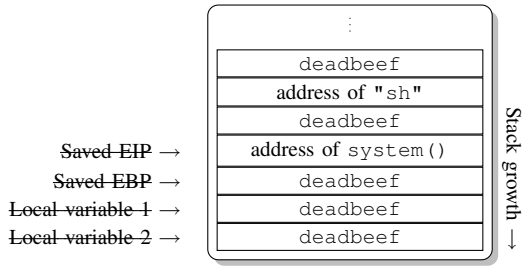

Fig. 2. Stack of the vulnerable process during a `return-to-libc` attack

4) A new protection that can be used to block our attack without recompiling the executables that incurs low overhead (about 2.69% with respect to the unprotected execution).

The remainder of the paper is organized as follows. Section II gives some background on the protections our attack bypasses. Section III presents the details of the attack. Section IV analyzes the effectiveness of existing solutions and presents a new technique to mitigate the attack. Section V evaluates our attack and the defense proposed. Section VI discusses the related work. Section VII concludes the paper.

II. Background

Figure 1 shows a sample program vulnerable to a traditional stack-based overflow. An attacker can exploit the buffer overflow to overwrite the stack with arbitrary data, thus forcing the program to execute arbitrary code. Modern operating systems mitigate this class of attacks with W⊕X, a policy that prevents memory pages containing executable data from being writable and vice versa. With such a policy in place, the only way for an attacker to execute arbitrary code is to mount a return-to-lib(c) attack [5], which consists of overwriting the return address of the vulnerable function and the following words of the stack with the address of a lib(c) function (e.g., `system`) and the arguments to pass to this function (e.g., the address of the string `"sh"`). Figure 2 shows the stack of the vulnerable process after the overflow, prepared to mount the return-to-lib(c) attack.

When address-space layout randomization (ASLR) is used in tandem with W⊕X, the attack becomes much more difficult. At every execution, the stack, the heap, and shared libraries (such as libc), are loaded at different random addresses. Consequently the attacker does not know the address of the function to return to. Nevertheless, Shacham *et al.* demonstrated that return-to-lib(c) attacks are still feasible in systems protected with address- space layout randomization using brute force [4]. The expected number of attempts for the brute force attack to succeed is 2^n, where n is the number of bits of randomness in the address-space. As an example, on GNU/Linux on IA-32 (where n is at most 16) 65,536 attempts are sufficient to successfully mount the attack.

III. Attack

A. Overview of the attack

Our attack against address-space layout randomization successfully returns to lib(c) in a single attempt, while Shacham *et al.*'s attack instead requires 2^n attempts (where n is the number of bits of the address-space subject to randomization). We propose an information leakage attack. Contrarily to the information leakage attack suggested by Durden [8], ours requires neither information about the current layout of the process, nor the ability to access arbitrary stack elements (e.g., to retrieve the address of `main()`). Instead, our attack exploits information about the base address of the lib(c), which is directly available in the memory of the process. The attack is built on an exploit technique [9], [10], that was previously thought to be inapplicable with ASLR. Our idea is to combine few code fragments that, despite ASLR, are available at absolute fixed addresses in the memory of the vulnerable process and to use these fragments to discover the base address of a dynamic library. Once the library has been de-randomized, we can return to any of its functions.

Figure 3 shows the layout in memory of our sample vulnerable program[1]. To ease the presentation, the layout is simplified: the stack and the heap are omitted and we assume that dynamic binding between the executable and the shared library has already been performed. The vulnerable program is loaded at address `0x8048000`. We assume the vulnerable program is compiled to be position dependent (that is the default compiler configuration) and consequently that its base address is fixed. This assumption is well-grounded because supported by empirical evidence: the large majority of executables found in modern UNIX distributions are not position independent (about 92.9%), only shared libraries are. The lib(c) instead is loaded at address `0xb7f50000`, but the address varies from execution to execution. The figure also shows the mechanism used to invoke the functions exported by the shared library, which is essentially an indirect jump table [11]. When the executable is loaded in memory, the linker transfers in memory all the requested libraries and then performs the relocation. The executable contains two special data structures (or sections) used specifically for the purpose of linking the executable with shared objects: the Global Offset Table (GOT) and the Procedure Linkage Table (PLT). The former is an array containing the address of the various library functions used by the program. The latter is an array of jump

[1]All the examples in the paper are specific for the x86 architecture, GNU/Linux (2.6.x), and ELF-32 executables. We use the AT&T assembly syntax.

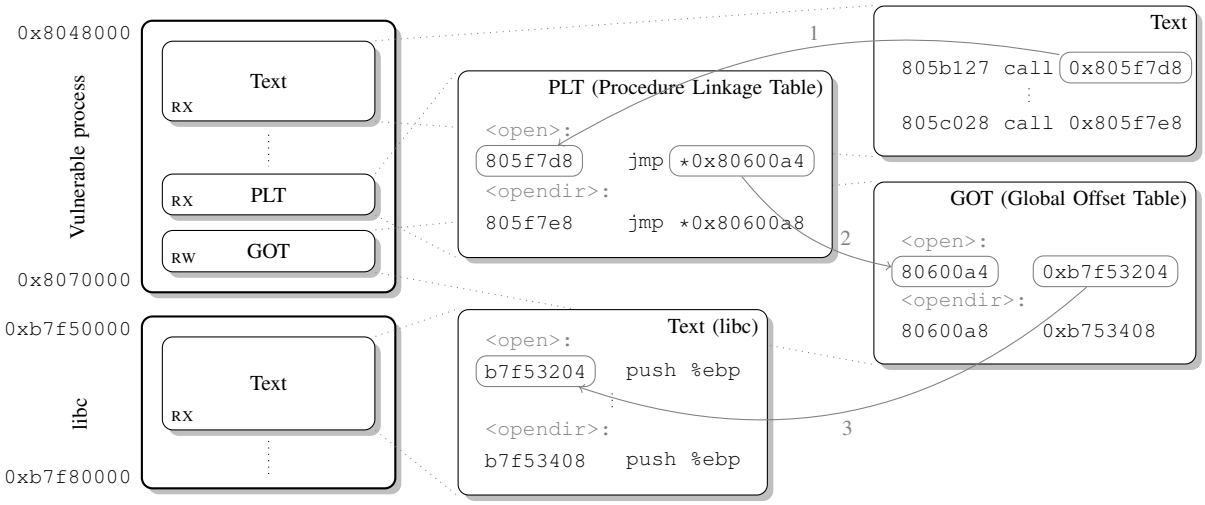

Fig. 3. Layout of a sample process and overview of the mechanism used to invoke functions residing in shared libraries

stubs. The i^{th} PLT entry contains a jump instruction that jumps to the address stored in the i^{th} GOT entry. The linker, at load time (assuming preemptive binding), fills the GOT with the addresses of the imported functions, updated to be consistent with the current base address of the library. The separation between PLT and GOT is for improved security: the former is executable but not writable, the latter is writable but not executable, thus preventing an attacker from writing and executing arbitrary code. For example, to invoke the libc function `open` our sample program performs a function call (instruction `0x805b127`), but instead of invoking a normal function, it invokes the stub for `open` in the PLT (located at address `0x805f7d8`). In turn, the stub of the PLT jumps to the code of the function inside the libc. The jump is indirect and the target of the jump is the address stored in the GOT entry of the `open` function (at address `0x80600a4`). In summary, through the call and the indirect jump the execution flows to `open` in the libc (in our sample process, the absolute address of `open` is `0xb7f53204`).

The knowledge of the absolute address of a single function exported by the lib(c) is sufficient to mount a successful attack, enabling any function in the library (including those not exported) to be invoked. Our attack exploits the information found in the GOT of the process to calculate the base address of the library, calculate the absolute address of an arbitrary function of the library, and subsequently invoke that function. Let $\mathit{offset}(s)$ be a function that computes the virtual offset, relative to the base address of the library, of the symbol s. It is worth noting that the virtual offset can be computed off-line from the library file and that the offset is constant. To ease the presentation, we use `open` to denote any function used by the attacker to de-randomize the library, and `system` to denote any function whose absolute address the attacker wants to compute. Given the absolute address of a library function, the base address of the library (`libc`) can be computed as follows:

$$\texttt{libc} \;=\; \texttt{open} - \mathit{offset}(\texttt{open})$$

Similarly, the absolute address of any function of the library can be computed as follows:

$$\texttt{system} \;=\; \texttt{open} - \mathit{offset}(\texttt{open}) + \mathit{offset}(\texttt{system})$$

Even though the math is trivial, it is very complex to perform in our context. Indeed, despite the stack overflow vulnerability, we cannot inject and execute our own code because the stack and all other data pages are not executable. A solution to overcome this limitation is to *borrow code chunks*, that is, to use code already available in the executable section of the process [9], [10]. Practically speaking, a code chunk is a sequence of bytes representing a sequence of one or more valid instructions that is terminated by a `ret` instruction. Although code chunks available are typically very simple and short, they can be combined, using *return-oriented programming*, by constructing powerful *gadgets*, i.e., short blocks placed on the stack that chain several code chunks together and that perform a predetermined computation [10]. An example of a code chunk is the string `8b 50 64 c3`, corresponding to the sequence of instructions `mov 0x64(%eax),%edx; ret`. The `ret` instruction ending each code chunk allows the construction of gadgets that link multiple chunks together. Figure 4 shows a sample stack configuration containing two gadgets that combine a 3-byte code chunk (the sequence of instructions `pop %eax; pop %edx; ret`) with another one, to read the content of arbitrary memory locations. During the overflow, the stack frames of the vulnerable function and the callers are overwritten with gadgets (see Figure 4). The first gadget starts exactly where the return address of the vulnerable function was stored before the overflow. It is composed of three double-words: the address of the first code chunk (`0x08055453`), and two integers (`0x8049167` and `0x80491a1`) that will be consumed during the execution of the code chunk. When the vulnerable function returns, the first code chunk is executed, and its execution results in the initialization of the two registers (i.e., `eax` and `edx`) with the values specified in the gadget (the second and third double-words of the gadget). The second gadget, being stored

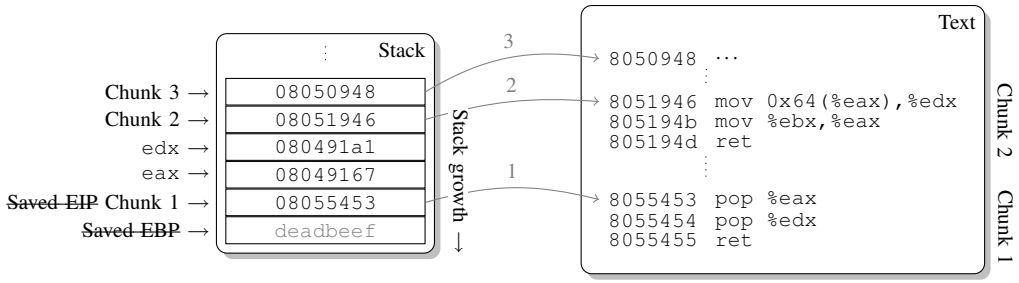

Fig. 4. Sample stack configuration with three gadgets, to chain the code chunks available in the vulnerable process

adjacently to the first one, causes the execution to flow from the first to the second code chunk. Indeed the `ret` instruction terminating the first code chunk references the double-word belonging to the subsequent gadget and representing the start address of the second chunk (`0x08051946`). The second code chunk reads the content of the memory location pointed by `eax` and stores the result in `edx`. Additional operations could be chained to perform more complex computations by writing other gadgets to the stack during the overflow.

The x86 architecture has a very dense and rich instruction set, instructions have variable length and do not need to be aligned. Therefore, code chunks are typically very frequent. However, those usable by an attacker are just a few. The numerous code chunks available in libc and other libraries cannot be used because of ASLR. As the executable is position dependent, only a few constant-address chunks in the code section can be used.

B. Details of the attack

Our attack uses the code chunks available in the code section of the vulnerable process to determine the base address of the lib(c), and uses this information to execute any function of the library. More precisely, our attack works as follows.

1) Identify the code chunks available in the vulnerable process.
2) Combine these code chunks to retrieve from the GOT of the vulnerable process the absolute address of a function of the lib(c).
3) Compute, again using the available code chunks, the absolute address of the function of the library we want to invoke.
4) Transfer the control of the execution to the latter function.

We present two variants of the attack. The first one is a straightforward application of the four steps described above. The second one has been developed to operate in situations where the first variant cannot, because the required code chunks are not available. The second variant indeed uses more common code chunks that allow to modify any entry of the GOT, without reading it explicitly.

1) Attack 1 – GOT dereferencing: The first attack combines gadgets to read the absolute address of any lib(c) function (e.g., `open`) from the GOT of the process, uses this address to compute the absolute address of another function of the library (e.g., `system`), and jumps to the address just computed. To do that we need the following gadgets: a load, an addition, and an indirect control transfer. Each of these gadgets can be obtained by combining one or more code chunks available in the code section of the vulnerable program. The x86 architecture facilitates the attack because it can perform complex tasks, such as a load and an arithmetic operation, with a single instruction. Therefore, the number of code chunks required to mount the attack is very small.

An example of a code chunk that constitutes one of the building blocks of our attack is the sequence of bytes `03 83 c4 5d 00 00 5f c3`, encoding the instructions `add 0x5dc4(%ebx),%eax; pop %edi; ret`. To turn such a code chunk into a dangerous gadget, it is sufficient to properly initialize the registers `eax` and `ebx`. Indeed, a proper configuration of the two registers enables to load the absolute address of `open`, and to compute the address of `system`. Let $got(s)$ be the address of the GOT entry of the symbol s. Like for the virtual offset of a symbol, the addresses of the various GOT entries of the program are constant, and can be computed off-line from the program file. The assignment to the two registers necessary to compute the absolute address of `system` is:

$$\begin{aligned} \texttt{eax} &= \mathit{offset}(\texttt{system}) - \mathit{offset}(\texttt{open}) \\ \texttt{ebx} &= \mathit{got}(\texttt{open}) - \texttt{0x5dc4} \end{aligned}$$

With this register configuration the instruction loads from the GOT the absolute address of `open` (the $-$`0x5dc4` delta is necessary because the instruction loads the data at address `ebx + 0x5dc4`) and sums it to the offset stored in `eax`. The result is saved in `eax`, and corresponds to the absolute address of `system`. To complete the attack, the attacker just needs a code chunk that transfers the execution to the address in `eax`. For example, the instruction `jmp *%eax` can be used for this purpose.

Figure 5 shows the stack of the sample vulnerable process during the attack and illustrates how the various code chunks are combined in gadgets by the attacker to perform the exploit. Overall, during the attack, the stack contains five different gadgets. The number can vary slightly, depending on the type of code chunks available in the vulnerable program[2]. The first code chunk (at address `0x8054126`) pops two double-words from the stack and stores them in `edi` and `ebp` respectively. The attacker uses the first gadget to initialize `edi` with the

[2]The gadgets used to illustrate the attack resembles the ones more common on GNU/Linux (x86) systems.

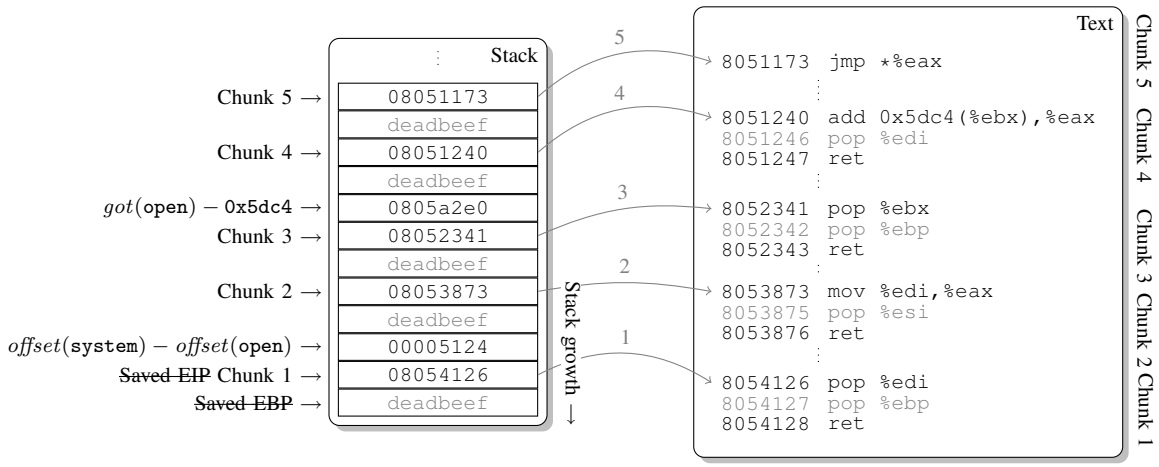

Fig. 5. Sample stack configuration for the GOT dereferencing attack, where the address of `system` is assumed to be `0xb7f58328` (instructions and elements of the stack irrelevant for the attack are shaded).

distance between `system` and `open`. The register `ebp` is irrelevant for the attack and its initialization is just a side effect of the code chunk. Indeed, the code chunk resembles a standard function epilogue, which restores callee saved registers. The `ret` instruction terminating the first code chunk triggers the second gadget, stored in the stack right above the element previously popped into `ebp`. The gadget uses the second code chunk (at address `0x8053873`) to copy the value of `edi` to `eax`. This operation is needed because we are assuming that no code chunks exists to directly initialize `eax`. Again, the `pop` instruction found in the chunk is a side effect. The third chunk (at address `0x8052341`) is used by the attacker to initialize `ebx` with the address of the GOT entry of `open`. The code fragment pops the value from the stack and saves in `ebx`. After the execution of the first three gadgets both `eax` and `ebx` are initialized as described earlier and the attacker has completed the preparation of the context for the execution of the gadget that computes the desired absolute address of `system`. The fourth gadget is used for the computation and to store the address in `eax`, and the fifth gadget is used to jump at the beginning of the `system` function, completing the attack.

2) Attack 2 – GOT overwriting: The second attack overwrites an entry of the GOT (e.g., the entry of `open`) with the address of another library function (e.g., the address of `system`), and transfers control to the selected function through the modified GOT entry. The attack is possible because, in the default setup, binding is performed lazily, and the GOT must be filled on demand. Hence, it must be writable.

The attacker needs the following gadgets: a load, an addition, a store, and an indirect control transfer (with a memory operand). Although apparently more gadgets are needed to perform this variant of the attack than to perform the previous one, in practice the first three operations can be performed using a single machine instruction; that is, an arithmetic operation with a destination memory operand, such as `add %eax,0x83d8(%ebx)`. This kind of code chunk is increasingly frequent in executables, relative to the type of chunk on which the GOT dereferencing attack is based. Furthermore, no particular control transfer instruction is requested to invoke the chosen library function as the PLT stub of the function whose GOT entry has been modified can be used for the attacker's purpose.

Figure 6 shows the stack of the sample vulnerable process during the attack, and illustrates how the various code chunks are combined in gadgets by the attacker to exploit the vulnerability. In total the attacker combines three gadgets, two of which perform two operations instead of a single one. The return address of the vulnerable function is overwritten with the address of the first gadget and the previous doubleword in the stack contains the distance between `system` and `open`. The first gadget (using the code chunk at address `0x8054341`) initializes the value of `ebx` with the distance stored in the stack. The second gadget (using the chunk at `0x8053123`) copies the value from `ebx` to `eax` and then initializes `ebx` with the address of the GOT entry of `open` that will be overwritten. The third gadget (using the chunk at `0x8052313`) computes the absolute address of `system`, as in the first attack, but with the operands in the inverse order, such that the computed address is stored directly in the dereferenced entry of the GOT. Finally, the `ret` instruction of the gadget is used to return directly to the PLT entry of `open`, in order to use its jump stub to invoke the function through the GOT.

IV. Attack mitigation

This section presents various protections mechanisms proposed in literature, and discusses their effectiveness at preventing our attack, when used in combination with W⊕X and ASLR. Furthermore, this section presents a new protection that can be used to block both variants of our attack.

A. Existing mitigation strategies

Table I reports the existing protections included in our analysis and compares them against the two variants of the attack we have developed. The highlighted row of the table is relative to the new mitigation technique proposed in the paper.

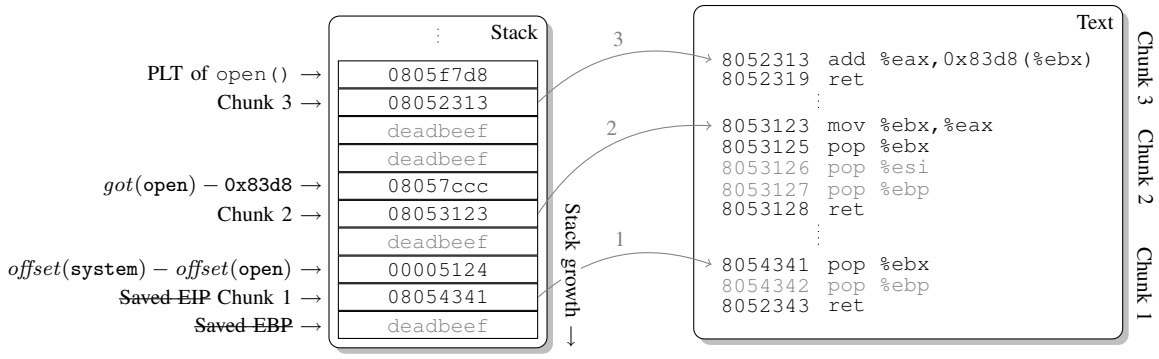

Fig. 6. Sample stack configuration for the GOT overwriting attack

Bhaktar *et al.* [12] proposed a randomization scheme that uses binary rewriting to periodically re-obfuscate an executable, including the layout of the code section. The randomization of the code section could prevent an attacker from using the chunks available in the executable. However, since re-obfuscation is periodic, a local attacker that can access the executable on disk can successfully mount both variants of our attack, within the time window in which the executable does not change. Xu *et al.* [13] designed a runtime system that randomizes the location of the GOT and patches the PLT accordingly. This system essentially just adds a fake layer of security: the sensitive information (the content of the GOT) is stored at a random location, but the address of this location remains accessible in memory (in the PLT). Through our attack it is possible to dereference the PLT, discover the address of the GOT, and then overwrite or dereference any GOT entry. However, to perform GOT overwriting, the code chunks necessary for a dereference must be available in the executable. Recent versions of `binutils` include support for producing executables with a read-only GOT [14]. A similar protection could also be implemented at runtime, by adopting a system like the one proposed by Xu *et al.* Clearly such protection prevents our GOT overwriting attack, but it cannot mitigate the first variant of the attack. Unfortunately, despite the fact that this protection has been available in `binutils` for years, our experimental analysis demonstrated that this protection is not yet adopted by any distribution (numbers are reported in Section V).

RedHat extended the idea of position independent code to executables. Like shared objects, position independent executables (PIE) can be loaded at arbitrary memory locations [7]. None of the variants of our attack can be applied to position independent executables because, as for randomized libraries, the address of code chunks varies from one execution to another. Therefore, guessing the absolute address of a code chunk in an executable becomes as hard as reusing the code of a shared library. To further complicate the exploitation, position independent executables can also be used to construct self-randomizing executables [15], executables that rearrange automatically, at each execution, the disposition of their functions.

B. Preventing unsafe accesses to GOT

Our attack is not possible on position independent executables for the aforementioned reasons. However, this feature is not yet widely adopted by modern UNIX distribution, but the motivations for such a choice are not clear (numbers are given in Section V). We speculate that vendors are afraid of the performance penalties PIE could introduce and are also not aware of its real importance. Although we strongly encourage vendors to move to position independent executables, we propose a new runtime solution that, being applicable without recompilation, can be used during the transition to PIE-enabled distributions and on operating systems where PIE is not yet available, but ASLR is (e.g., OpenBSD).

Our solution is inspired by the randomized GOT protection proposed by Xu *et al.* [13], and relies on *encrypting the content of the GOT*. The idea is to encrypt GOT entries, to prevent all but legitimate accesses. With the exception of the accesses performed by the dynamic linker to bind the executable with the shared libraries, all further accesses to the GOT are reads and occur only from the PLT (see Figure 3). Therefore, besides the linker, only the accesses originating from the PLT should be considered legitimate and authorized to access to unencrypted content of the GOT and to transfer the execution to the functions in shared libraries. To ease the presentation we assume preemptive binding (i.e., `LD_BIND_NOW` is set). In such a situation all legitimate accesses to the GOT originate from the PLT. However, the approach we are proposing could be extended to work also with lazy binding, by customizing the dynamic linker.

In more detail, our scheme operates as follows. We encrypt all the entries of the GOT such that attacker's attempts to read the content of the GOT to guess the random base address of the library fail; without the decryption key, the retrieved content of the GOT is meaningless. Similarly, attempts to modify the GOT fail as well. Obviously, encryption interferes with the correct execution of the program. For this reason, we rewrite the program to make it able to decrypt the protected data when it legitimately accesses the GOT. As all legitimate accesses go through the PLT, it is sufficient to patch each stub of the PLT to dereference and decrypt the corresponding GOT entry, and then to transfer the execution to the decrypted address. The weakness of the randomized GOT protection proposed

	GOT dereferencing	*GOT overwriting*	*Requires recompilation*
W⊕X and ASLR	–	–	No
Periodic re-randomization [12]	–	–	Yes
GOT randomization [13]	–	–	No
GOT read-only [14]	–	✓	No
PIE [7]	✓	✓	Yes
Self-randomization [15]	✓	✓	Yes
Encrypted GOT	✓	✓	No

TABLE I
COMPARISON OF EXISTING PROTECTIONS WITH RESPECT TO OUR ATTACK AND TO THE NEW PROPOSED PROTECTION (✓ DENOTES THAT THE DEFENSE TECHNIQUE PREVENTS THE ATTACK)

by Xu *et al.* is that the PLT leaks the address of the GOT, and consequently an attacker can mount both a GOT dereferencing and GOT overwriting attacks. As we adopt a similar strategy, we are exposed to the same risk. Therefore, we have to protect the patched PLT to avoid any information leak that can be exploited by the attacker.

Each PLT entry is patched to perform the following operations: (I) to read the corresponding GOT entry, (II) to decrypt the address read, (III) and to jump to the decrypted address. Because of the aforementioned problem, the decryption key cannot be stored directly in the code of the patched jump stub, nor can it be referenced explicitly from the code. The solution we adopt inlines in the i^{th} jump stub a key generation function that computes dynamically the decryption key to use for the decryption of the i^{th} entry of the GOT. Encryption keys and key generation functions are generated at runtime and differ from one GOT/PLT entry to another. The rationale behind this choice is that, although an attacker could read (using our GOT dereferencing attack) the code of the patched jump stub that performs the decryption, and try to "borrow" the decryption code, he does not know how to use this code. Indeed, this code is generated randomly at each execution, and to construct useful gadgets from it the attacker would have to analyze (i.e., disassemble) the code, and the only way to do that is to use other gadgets. Although that is theoretically possible, such a complex analysis requires an arsenal of gadgets that are practically impossible to construct even from a large executable.

We have developed a prototype implementation of the proposed protection, for GNU/Linux (x86). For simplicity the prototype requires preemptive binding of shared libraries and does not support dynamic loading of shared objects (e.g., `dlopen`). However, the dynamic linker could be extended to support our protection also with lazy binding and dynamic library loading. Lazy binding typically introduces less overhead and reduces startup costs, and consequently the overhead introduced by our protection could be reduced by completing the prototype. Our prototype consists of a shared library that is injected in the address space of the program to protect (using `LD_PRELOAD`). The library encrypts all the entries of the GOT and then patches the jump stubs of the PLT as described above. Since the size of PLT entries is insufficient to hold our patched code and cannot be enlarged without breaking the functionality of the program, we allocate a new executable section, and store in it the new patched entries. Additionally, we update PLT entries to redirect the execution to the corresponding entries in PLT_ENC. Keys generation functions are constituted of a random number (up to a dozen) of different assembly instructions, and are crafted to be unpredictable. Figure 7 shows the memory layout of our sample process with the randomized GOT protection in action. The extra section called PLT_ENC is the section created to hold the new encryption-aware jump stubs. When the program calls the `open` function, the execution flows, through the patched PLT, to the jump stub of `open` in the PLT_ENC section (at address `0x08078177`). The code we use to decrypt the GOT entry of `open` and then to invoke the function looks like the code in the figure, but it is different in each entry and execution. The code performs the three operations necessary to invoke the function (load, decryption, and control transfer) and takes the precaution of preserving all registers.

V. EVALUATION

We evaluated the proposed attack and solution. Overall, the evaluation demonstrated the wide-scale applicability of our attack, and the effectiveness of the proposed protection. Details of the evaluation are reported separately in the following sections.

A. Evaluation of the attack

We performed two independent evaluations for our attack. First, we tested our attack against a version of Ghostscript vulnerable to a stack-based overflow. We successfully exploited the vulnerable program with both variations of our attack. Second, we tested a large corpus of programs, collected from different UNIX distributions for the x86 and x86-64 architectures and supporting both W⊕X and ASLR, to measure how many of them were *predisposed to the attack* (i.e., whether the attack would be possible if the programs were vulnerable to a stack-based buffer overflow). For the x86 architecture, the majority of the programs tested, about 95.6%, were found to be predisposed to the attack. For the x86-64 architecture we found less predisposed programs, only about 61.8%. This is due to the fact that on x86-64 instructions with 64-bit operands requires a special prefix, and consequently the code chunks needed for the attack are less frequent.

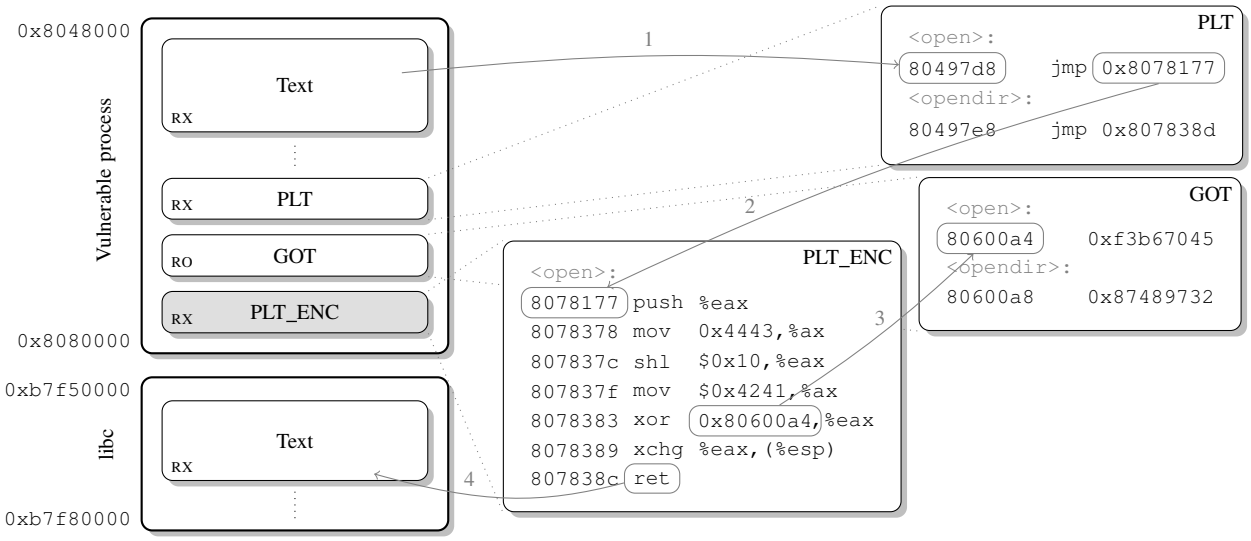

Fig. 7. Layout of the sample vulnerable process with our of GOT encryption protection enabled

1) Automation of the attack: For the evaluation we have implemented a prototype tool, called SARATOGA, that automatically analyzes a ELF-32 or ELF-64 executable (for x86 and x86-64 architectures respectively), detects whether the program is predisposed to any of the two variations of the attack, and generates a stack configuration that can be used to exploit a vulnerability in the program. To find code chunks in an executable, SARATOGA uses the algorithm presented by Shacham *et al.* [10]. SARATOGA combines available code chunks using a custom algorithm we have developed. Our algorithm is goal-oriented and rule-based. Given a code chunk that either allows to dereference or to overwrite a GOT entry, the algorithm assigns a predetermined value to each possible use of the code chunk (e.g., the source operands of the instructions in the chunk). The algorithm uses a set of combination rules and tries to apply them recursively, to combine the available code chunks to perform the requested assignments. If multiple combinations are possible, the algorithm selects the one consuming less stack space. For output, SARATOGA produces a stack configuration containing the gadgets for the exploitation.

2) Exploiting a real vulnerability: We tested our attack against a vulnerable version of Ghostscript [16]. We initially developed a conventional exploit and tested it against the program with W⊕X and ASLR disabled. The exploit worked correctly and gave us a shell. Subsequently, we enabled the two protections and verified that the exploit stopped working. We run SARATOGA on the image of the program under attack and, in few seconds, obtained two stack configurations for the two variants of the attack. We constructed two exploits using the results provided by SARATOGA, and successfully exploited the vulnerability and obtained a shell with both.

3) Wide-scale applicability of the attack: The evaluation targeted the executables found in the directories `/bin`, `/sbin`, `/usr/bin`, and `/usr/sbin` of the following distributions: GNU/Debian "Squeeze" (x86), GNU/Debian "Lenny" (x86-64), Fedora "Cambridge" (x86), OpenBSD 4.5 (x86-64). For the total set of executables found in each distributions we selected for the evaluation only those whose code size was greater than 20Kb. The rationale was that excessively small executables have a limited functionality and very seldom attract attackers. On the contrary, commonly-attacked executables (e.g., Ghostscript, Samba, Apache) are bigger, of the order of tens or hundreds of kilobytes.

The results of our evaluation are reported in Table II. For each distribution, the table reports the total number of executables analyzed, the percentage of position dependent executables, the percentage of executables with writable GOT, the percentage of executables vulnerable to the GOT dereferencing attack, the percentage of executables vulnerable to the GOT overwriting attack, and the percentage of executables vulnerable to any of the two attacks. All three tested Linux distributions support PIE and non-writable GOT. Unfortunately, our results testify that these mitigation techniques are not yet widely used. As the table shows, the large majority of the executables for x86 are predisposed to at least one of the two variants of the attack. The second variant has much larger applicability, because the requested code chunks are more common. The attack is not as effective on x86-64 executables, but still, more than half of the tested executables are predisposed to it. With the exception of OpenBSD executables (where PIE is not available), all the executables found in the other distribution would not be predisposed to the attack if they were PIE. Furthermore, considering that the number of programs predisposed to the GOT dereferencing attack is much smaller that the percentage of programs predisposed to at least one of the two attacks, the read-only GOT protection would give non-negligible benefits.

It is worth noting that the a vulnerability found in an executable predisposed to our attack might not be exploitable. For example, the vulnerability might not expose a large enough portion of the stack, or it might not provide the needed stack manipulation operations (e.g., to inject null bytes). These

	Debian (x86)	*Debian (x86-64)*	*Fedora (x86)*	*OpenBSD (x86-64)*
Executables	509	333	590	174
non-PIE	95.7%	97.3%	85.8%	100%
Writable GOT	99.8%	100%	99.0%	100%
Attack 1	64.0%	17.8%	49.5%	58.6%
Attack 2	96.1%	57.4%	95.0%	68.4%
Any attack	96.3%	58.3%	95.0%	68.4%

TABLE II
EXPERIMENTAL EVALUATION OF THE EFFECTIVENESS OF THE ATTACK ON X86 AND X86-64 EXECUTABLES

	`bc`	`bogofilter`	`bzip2`	`clamscan`	`convert`	`grep`	`oggenc`	`tar`	*Avg.*
PIE	10.55%	3.46%	0%	0.12%	0%	1.41%	0.16%	0.12%	1.98%
Encrypted GOT	0.21%	15.49%	0.63%	0.11%	0.32%	4.54%	0.02%	0.20%	2.69%

TABLE III
OVERHEAD INTRODUCED BY PIE AND BY OUR PROTECTION (THE BASELINE FOR COMPARISON ARE THE NON-PIE EXECUTABLES)

situations are not considered a limitation of our attack but rather a limitation of the vulnerability itself.

B. Evaluation of the proposed defense

We evaluated the efficacy our encrypted GOT protection, as well as the overhead it imposes. Our results demonstrate the effectiveness of our solution at stopping both attack variations, as with a small runtime overhead (about 2.69%).

To evaluate the effectiveness of the proposed mitigation strategy we tested the two exploits constructed with the help of our tool against the vulnerable version of Ghostscript, with our GOT protection, W⊕X, and ASLR enabled. Both exploits failed to work. The vulnerable process terminated with a page fault exception caused by an access to an invalid memory page.

We evaluated the overhead introduced by our protection and compared with the overhead introduced by PIE. For the evaluation we used the following applications: `bc`, `bogofilter`, `bzip2`, `clamscan` , `convert`, `grep`, `oggenc`, and `tar`. These applications are CPU-bound and make frequent use of functions in shared libraries. Experiments were performed on an x86 system running GNU/Linux 2.6.27. As our protection works entirely in user-space, for each application we measured the user-time requested to complete a batch job, averaged over multiple runs, in three different configurations: (I) with a version of the executable compiled with default options (position dependent executables), (II) with a version of the executable complied with the default options as PIE, and (III) with the first version of the executable but with our runtime protection enabled. Table III reports the overhead measured with each application and the average. The percentages in the table represent the relative increment of user-time, with respect to configuration (I). From these results we can draw two main conclusions. First, the average overhead introduced by PIE is very small, 1.98%, and a maximum of 10.55% with `bc`, and can be further reduced with more aggressive compilers optimizations (e.g., by omitting the frame pointer). Second, the overhead introduced by our encrypted GOT protection is also very small and comparable to that introduced by PIE. The average overhead observed was 2.69% and a maximum of 15.49% with `bogofilter`, which invokes library functions with a very high frequency. The small overhead implies practical adoption of our protection on both end-users and production systems.

VI. RELATED WORK

Since the traditional technique for exploiting stack-based buffer overflows was first disclosed [17], several other exploiting techniques have been invented and classes of vulnerabilities have been discovered. The techniques mostly related to our work have been already presented in great detail in the main sections of the paper. The most important vulnerabilities to mention instead are heap-based overflows [18], format string vulnerabilities [19] and integer overflows [20].

We observed a parallel development of techniques to protect applications from memory error exploits. In Section IV we discussed the main attack mitigation strategies related to our work, such as W⊕X, ASLR, the various protections extensions to ASRL and the protections for the GOT. Aside from these techniques, other approaches have been proposed. StackGuard [21] is a compile-time solution that protects programs from stack-based overflows by placing canary values between a function's local variables and its return address. Canaries are used to detect corruptions of the stack. ProPolice extends StackGuard by reordering functions and local variables and relocating code pointers in the stack items preceding dangerous buffers to impede overwrites [22]. PointGuard uses encryption to protect pointers [23]. As in the protection for the GOT we propose, pointers are decrypted at dereference time. StackShield uses a shadow stack to save the return addresses and to check that they have not been tampered at function returns. For a survey of traditional mitigation techniques, the interested reader can refer to [24].

A completely different approach to detect memory corruptions is the N-Variant system [25]. The idea is to run n different instances of the same program with diverse memory layouts, obtained using ASLR. Any successful attack will work only on one of the instances and will cause all the other instances to crash because the attack must be tailored to a particular process layout. This idea has been further extended in [26].

VII. Conclusions

We presented a new attack against programs vulnerable to stack-based buffer overflows that bypasses two of the most widely adopted protection techniques, namely write or execute only (W⊕X) and address space layout randomization (ASLR). With our attack we extract from the address space of the vulnerable process information about the random base address at which a library is loaded and then use this information to mount a return-to-lib(c) attack. Contrary to the state-of-the-art attack for this scenario, which is based on brute-force and requires a number of attempts proportional to the size of the address space, using our attack the vulnerable program can be subverted in a single shot. We analyzed the executables found in different UNIX distributions and observed that the attack would be successful on the majority of them. We also analyzed several existing protections that can be combined with W⊕X and ASLR. Our finding was that ASLR is effective against our attack only when used in combination with position independent executables (PIE). Unfortunately the latter approach is not yet very popular and requires recompilation. We proposed a new runtime protection that impedes our attack without having to recompile programs and introduces a small overhead.

References

[1] E. H. Spafford, "The Internet Worm Program: an Analysis," *SIGCOMM Computer Communication Review*, vol. 19, no. 1, pp. 17–57, 1989.

[2] The PaX Team, "PaX non-executable pages." [Online]. Available: http://pax.grsecurity.net/docs/noexec.txt

[3] ——, "Address space layout randomization." [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[4] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-Space Randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, 2004, pp. 298–307.

[5] S. Designer, ""return-to-libc" attack," Bugtraq, 1997.

[6] "grsecurity." [Online]. Available: http://grsecurity.net

[7] A. van de Ven, "New Security Enhancements in Red Hat Enterprise Linux v.3, update 3," Aug. 2004.

[8] T. Durden, "Bypassing PaX ASLR protection," Jul. 2002.

[9] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," 2005.

[10] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2007, pp. 552–561.

[11] J. Levine, *Linkers and Loaders*. Morgan Kaufmann, 2000.

[12] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits," in *Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 105–120.

[13] J. Xu, Z. Kalbarczyk, and R. Iyer, "Transparent Runtime Randomization for Security," University of Illinois at Urbana-Champaign, Tech. Rep. UILU-ENG-03-2207, May 2003.

[14] "GNU `binutils`," http://www.gnu.org/software/binutils/.

[15] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," in *Proceedings of the 14th USENIX Security Symposium*, Aug. 2005.

[16] CVE-2008-0411, "Ghostscript zseticcspace() Function Buffer Overflow Vulnerability."

[17] Aleph One, "Smashing The Stack For Fun And Profit," *Phrack Magazine*, vol. 7, no. 49, 1996.

[18] Michel Kaempf, "Smashing The Heap For Fun And Profit," *Phrack Magazine*, vol. 11, no. 57, 2001.

[19] Scut, Team Teso, "Exploiting Format String Vulnerabilities," March 2001.

[20] blexim, "Basic Integer Overflows," *Phrack Magazine*, vol. 11, no. 60, 2002.

[21] 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 *Proceedings of the 7th USENIX Security Symposium*, 1998, pp. 63–78.

[22] H. Etoh, "GCC extension for protecting applications from stack-smashing attacks (ProPolice)," 2003. [Online]. Available: http://www.trl.ibm.com/projects/security/ssp/

[23] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities," in *Proceedings of the 12th Usenix Security Symposium*, 2003.

[24] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings of the DARPA Information Survivability Conference and Exposition*, Jan. 2000, pp. 11–19.

[25] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-Variant Systems: A Secretless Framework for Security through Diversity," in *Proceedings of the 15th USENIX Security Symposium*, 2006, pp. 105–120.

[26] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified Process Replicae for Defeating Memory Error Exploits," in *Proceedings of the 3rd International Workshop on Information Assurance*, 2007, pp. 434–441.