Retrospective Protection utilizing Binary Rewriting

Sergej Proskurin, Fatih Kilic, Claudia Eckert[1]

Abstract:
Buffer overflow vulnerabilities present a common threat. To encounter this issue, operating system support and compile-time security hardening measures have been introduced. Unfortunately, these are not always part of the shipped object code. We present design and implementation of BinProtect, a binary rewriting tool, capable of retrospectively protecting binaries, which have not been sufficiently secured at compile-time. To achieve this, we do not need source code or any additional information.

1. Introduction

The stack buffer overflow, aka. the stack smashing attack [1], presents presumably the most known and one the most dangerous attacks on software applications. One of the first worms in history of computer security (the Morris worm [2]) exploited already in 1988 a stack buffer overflow vulnerability. And even today – almost three decades later – a buffer overflow presents one of the most feared attacks. Figure 1 illustrates the course of reported buffer overflow vulnerabilities[2]: Despite the trend concerning buffer overflows is slowly going back, it remains a common threat.

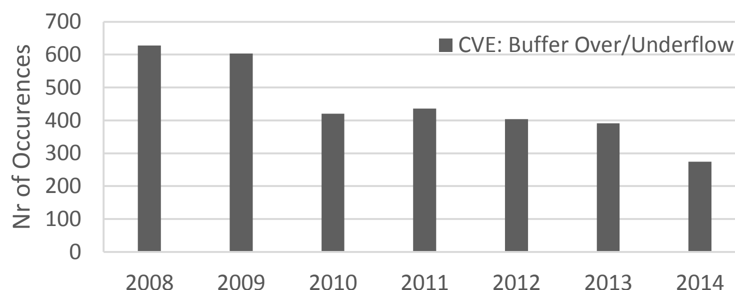

Figure 1: Buffer overflow related CVE entries, reported since 2008.

Buffer overflow vulnerabilities in general are the result of insufficient array bounds checks by the compiler or the applications themselves. Precisely this lack enables the attacker to overwrite function's return addresses and hence take over control of the associated process and its privileges. To limit this attack vector, special compile- and link-time security measures have been introduced [3, 4, 5, 6]. Unfortunately, these are not always part of the shipped object code. To encounter the issue of unprotected and potentially vulnerable object code, this paper presents *BinProtect*, a static binary protection mechanism, applying modifications on the Executable and Linking Format (ELF) [7] and post link-time object code transformation techniques, provided by Dyninst [8] and PatchAPI [9]. BinProtect modifies binaries with the objective of mitigating potential buffer overflow vulnerabilities and consequences by statically integrating security measures, which have been disabled at compile-time. This way, BinProtect retrospectively fortifies binaries and hence timely decouples development and compilation from the security hardening process.

1 Technische Universität München
2 CVE: http://cve.mitre.org/data/downloads/allitems.csv, online Jan 2015.

The paper is outlined as follows: Section 2 shortly relates BinProtect to similar work. Section 3 introduces general security mechanisms, enabled by the compiler and partly enforced by the Linux kernel. Inspired by the presented security mechanisms, section 4 and 5 outline the design and implementation details of their retrospective incorporation into binaries as it is done by BinProtect. We evaluate our prototype in section 6 and finally conclude this paper with references to future work in section 7.

2. Related Work

A binary rewriting Return Address Defense (RAD) has been presented by Prasad and Chiueh [10]. RAD introduces a binary rewriting engine that retrospectively fortifies binary objects for Windows platforms. Similar to BinProtect, RAD utilizes a shadow stack to temporarily store return addresses of stack frames at function entry and match these values with actual return addresses at function exit. In contrast to BinProtect, RAD is restricted to mitigation of buffer overflow vulnerabilities on the stack.

SecGOT [11] randomizes the address of the Global Offset Table (GOT) at load-time and hence decreases the chances of GOT entry modifications. BinProtect, on the other hand, tries to completely eliminate the possibility of GOT tampering by preventing the lazy binding mechanism of the dynamic linker and marking the GOT as read-only.

In their paper, Bernat and Miller present capabilities of PatchAPI [9] by closing three vulnerabilities in a running Apache server. Our implementation utilizes PatchAPI but in contrast to the prototype presented by Bernat and Miller, we do not concentrate on already reported vulnerabilities but rather fortify binaries in a prophylactic manner.

3. Binary Protection

For buffer overflow attacks to be successful, several conditions must be met: First, an attacker requires the presence of vulnerabilities. Second, he needs to modify control flow structures. Finally, to take over control, an attacker must locate and execute previously injected malicious code. Aiming at prevention of the upper conditions to become true, this section presents core methodologies that may be enabled by the *GNU Compiler Collection* (GCC) and partly enforced by the Linux kernel. These methodologies present fundamental concepts, which, if not available, are retrospectively integrated into binary objects by BinProtect as discussed in section 4 and 5.

3.1 Execution Prevention: NX-Bit

In general, every ELF segment that can be written to at run-time presents a potential vulnerability. One cannot rule out that an attacker may place malicious code or data into one of the writable memory regions (usually part of data segment) and hence influence the process for his benefit. Naturally, to preserve modern program compatibility, it is not always feasible to prohibit execution within the entire data segment[3]. For instance, applications and libraries utilizing certain programming constructs, such as dynamic code generation, may require

3 Former Unix and Windows systems utilized segment-wide access permissions, strictly separating executable code from writable data and stack segment.

the data segment to be executable. However, the stack segment certainly should be protected from malicious code execution. To counter buffer overflow attacks, attempting to execute malicious code on the stack, AMD introduced a memory execution prevention technology: The *No eXecute* (NX) bit. In contrast to coarse-grained and mostly inflexible protection provided by segmentation, the NX-bit enables fine-granular execution prevention on a per page basis. Thus, page table entries associated with the stack segment may be marked as non-executable. GCC achieves just that by advising the linker with the option *–z noexecstack* to insert the program header *PT_GNU_STACK* into the ELF file, indicating a non-executable stack. Yet, in special cases, both GCC and Linux require the stack to be executable: Former Linux kernels handle signals directly on the stack and GCC requires an executable stack within the context of nested functions. In addition, in case the *dynamic linker*[4] determines that one of the *Dynamic Shared Objects* (DSO) requires an executable stack, it will ignore permissions provided by the *PT_GNU_STACK* program header and enable its execution.

3.2 Stack Protection

There exist two types of stack overflow protection mechanisms: stack overflow protection based on *bounds checking* and *integrity checking*. Bounds checking approaches try to completely eliminate the issue of buffer overflows by tackling the problem by its source: These approaches try to prevent buffer overflows by implementing a mechanism, which checks the bounds of arrays, whenever they are accessed. This check can be a part of the compiler [12] or the application itself. However, because of potential dramatic performance degradation and compatibility preserving issues, bounds checking approaches are not considered very practical. Integrity checking based mechanisms, on the other hand, provide the possibility to detect compromised activation records (stack frames) by either inserting a terminator value, which is referred to as *canary*, directly into the stack frame [3], or saving the associated return address within a dedicated memory region [4, 6, 10]. In both cases, the integrity of activation records is validated by checking the canary or the redundantly stored return address with the associated value on the stack. In case the value was changed, the integrity of the associated activation record has been compromised. GCC enforces stack protection with *-fstack-protector(-all|-strong)* flags implementing a canary-based approach.

3.3 Standard C Library Consolidation

Besides the lack of array bounds checking mechanisms in languages, such as *C* and *C++*, the risk of a potential buffer overflow vulnerability arises when using certain functions provided by the *standard C library* (libc). Functions, such as *strcpy()*, *gets()*, *sprintf()*, and more, are classified as unsafe and rely on the programmer to make sure that they do not fill buffers beyond their bounds. To mitigate this risk, *Libsafe* [9] has been introduced as a layer between the binary

4 The dynamic linker, also referred to as the dynamic loader, is responsible for making the program ready for execution. It is implemented as a shared object so that it may be shared between ELF files.

and the dynamically linked standard C library. This layer comprises wrappers being able to intercept and harden calls to unsafe libc functions.

Similar to Libsafe, the GCC compiler, supplies the flag *–D_FORTIFY_SOURCE* that performs bounds checks on arrays and pointer references in connection with a number of unsafe libc functions. The GCC compiler benefits from the fact of having sufficient information to perform code modifications at compile-time and hence, in contrast to Libsafe, hardens calls to the libc not only when it is dynamically but as well statically linked against the binary object to be protected.

3.4 Global Offset Table Protection

Position Independent Code (PIC) and *globally shared libraries* introduce an indirection step, required to access the *Dynamically Shared Objects* (DSO). In contrast to the earlier concept of *load-time relocation* of shared libraries [13], Position Independent Code utilizes a *Global Offset Table* [13], containing pointers to global objects. This way, instead of relocating every reference to a shared object within the *.text* segment, PIC limits the amount of relocations to the number of entries within the GOT. At compile-time, addresses of shared objects are unknown and hence they need to be resolved or *relocated* by the dynamic linker at load- or run-time. Relocations, specifying memory addresses to be modified, are announced within the sections *.rel.dyn* and *.rel.plt* of the ELF file. Thus, the dynamic linker is able to identify memory addresses to be updated with addresses to the particular DSOs. Inside the ELF file, the GOT is usually distributed across the sections *.got* and *.got.plt*, both residing at fixed offsets within the *.data* segment. The section *.got* contains references to global data to be relocated at load-time, whereas the section *.got.plt* comprises references to global position independent functions to be relocated at run-time. To reduce overhead at program startup, external functions are not immediately relocated. Ergo, an additional table is incorporated into the *.text* segment of the ELF file: The *Procedure Linkage Table* (PLT). The PLT consists of trampolines to *.got.plt* entries, containing addresses of external functions. The PLT allows the dynamic linker not to relocate external functions until they are called for the first time. This process of delayed address resolution is referred to as *lazy binding*.

Since the dynamic linker updates the *.got.plt* part of the GOT at run-time, the associated segment needs to remain writable and hence vulnerable: A potential attacker may manipulate the process' control flow by modifying the GOT table entries. The same applies to entries of the sections *.init_array*, *.fini_array* etc., containing addresses of functions to be executed at program startup and end. To reduce the attack vector, GCC offers the *RELRO* mechanism making the linker mark the above listed sections as read-only after load-time relocations have been performed. The linker options *-Wl,-z,relro* and *-Wl,-z,relro,-z,now* distinguish between *partial* and *full RELRO*. Partial RELRO makes the linker arrange the sections listed above that are populated at load-time in a way that the dynamic linker may subsequently mark the associated pages as read-only. The part of the data segment, whose permissions are changed in this way, is specified by the

information held within the *PT_GNU_RELRO* program header. Nonetheless, because the *.got.plt* is bound lazily, it remains writable at run-time. In contrast, full RELRO resolves all DSOs at load-time. It therefore arranges the sections in a way that the entire GOT may be marked as read-only.

4. Design of BinProtect

The past has shown that the development of highly secure systems is very hard. Security hardening measures need to be utilized in every abstraction layer beginning from the logic and circuit design to hardware architecture and software applications. Security flaws in one of the abstraction layers has the potential to shatter the security of others. The idea behind BinProtect is to reduce the possibility of a successful attack on software applications. BinProtect allows to inject security hardening mechanisms into weakly protected object-files compiled for Linux platforms, to enable similar protection as provided by modern compilers. Individual mechanisms that are presented in section 3 may be retrospectively incorporated into shipped applications. To achieve this, our prototype classifies the presented hardening measures based on two binary transformation concepts required for their implementation: *binary editing* and *ELF transformation*. Both, the classification of the enforced protection mechanisms and the associated binary transformation methodologies are shortly described in the following.

4.1 Binary Editing

BinProtect applies static object code modification techniques, utilizing capabilities provided by the analysis and instrumentation framework Dyninst [8] and the binary code patching library PatchAPI [9]. The comprehensive static analysis of Dyninst allows a precise deduction of function and control flow from binaries without the need for any debugging information – even in stripped and highly optimized form [9]. PatchAPI utilizes the gathered static analysis information to perform fine granular object code transformations. In addition, PatchAPI introduces the concept of *structured binary editing* [9], comprising a transformation algebra to ensure validity of the instantiated binaries. This technique views the binary from a higher, more abstract level: The binary is not considered any more as a collection of subsequent instructions but as a flow of basic blocks interconnected by edges in form of a *Control Flow Graph* (CFG). Thus, transformations of binaries are performed on the CFG level. To preserve structural validity of binaries, Bernat and Miller introduce constraints on CFG transformations in form of a *transformation algebra* [9]. The CFG transformation algebra transforms the underlying control flow of the associated binary, and allows to insert so called *snippets* in form of a special abstract syntax language or raw binary code into the basic blocks of the previously transformed CFG. This way, the original binary may be extended and its control flow transformed, while simultaneously preserving its validity. Based on these concepts, BinProtect rewrites binaries to integrate mechanisms, required to protect the stack and fortify calls to insecure functions of libc, as described in sections 3.2 and 3.3.

4.1.1. Standard C Library Consolidation

BinProtect provides safe versions of the known set of unsafe standard C library functions and rewrites the binary in a way that all calls to unsafe functions in question are intercepted. Therefore, it needs to distinguish between whether the libc has been statically or dynamically linked against the binary to be protected and modify the binary accordingly. For this, BinProtect applies Dyninst's binary rewriting capabilities. Hence, the modified binary obtains hardened calls to both dynamically and statically linked standard C library functions. Similarly to Libsafe [6], for the implementation of the unsafe standard C library functions, our idea is to associate buffers with stack frames and make sure that they do not spill over return addresses. Thus, control flow information remains protected but general buffer overflows are not entirely eliminated.

4.1.2. Stack Protection

BinProtect collects sufficient static function and control flow information from binaries by means of extensive static binary analysis capabilities of the underlying framework Dyninst. Based on the statically gathered information, BinProtect extracts functions that are not part of shared libraries and subsequently performs static object code modification in respect to structural validity of the binary object by applying the transformation algebra introduced by PatchAPI [9]. The general idea behind the stack protection mechanism of BinProtect is to modify the extracted functions so that they become capable of performing integrity checks of the associated activation records at run-time. As a result – in contrast to GCC *-fstack-protector* variants, which implement a canary-based approach – the binary becomes able to check integrity of its activation records by utilizing a shadow stack mechanism: Therefore, the binary stores return addresses of active functions within a dedicated memory region during the function's prologue and matches the associated return addresses with their copies on the shadow stack during the function's epilogue. Thus, BinProtect detects stack buffer overflows and makes sure that the attacker does not take over control.

4.2 ELF Transformation

The Executable and Linking Format [7] defines a common object file format used in Unix-based operating systems for executable-, relocatable-, and shared object files. The ELF header acts as a guide, helping to locate essential components within the rest of the file. An ELF file may further comprise a *section header table* and a *program header table*, representing two different views on the ELF file: The *linking-* and the *program execution view*. The section header table keeps track of all *sections* containing essential data required for the process of linking. The program header table distributes all sections across *segments*, which are loaded into memory considering specified permissions. Thus, the ELF format determines the way how the file is handled at load- and run-time. This means, modifications of ELF files directly influence its future behavior. The ELF format is often abused for malicious purposes. Both, kernel and user space applications may suffer from so called *ELF infections* [14, 15], which hide inside proper

applications abusing their hospitality. Within the context of BinProtect, we apply such ELF infections or rather transformations with the purpose of security hardening binaries by mechanisms, which prevent execution on the stack and provide protection of the entire GOT, according to sections 3.1 and 3.4. The following presents ELF transformations required to incorporate these protections.

4.2.1. Execution Prevention: NX-Bit

Our idea is to modify ELF files in a way that the kernel (assuming sufficient support) will mark pages of the stack segment as non-executable. Therefore, we insert the program header of type *PT_GNU_STACK* into ELF executables in case of its absence or otherwise make sure it has no execution permissions. This does not alter the process' behavior but cause the kernel to mark the affected pages as non-executable. This technique does not directly prevent or recognize buffer overflows but rather alleviates their effects: Execution attempts of previously injected malicious code on the stack will be aborted. As a consequence, a number of stack buffer overflow vulnerabilities may be mitigated. Depending on the policy of the underlying operating system, heap-based buffer overflows may as well be eliminated respectively. Attacks, such as return-to-libc [16], however, remain possible. Permissions specified by the *PT_GNU_STACK* header are ignored if the binary makes use of libraries requiring an executable stack. However, programs that make use of an executable stack must skip this step.

4.2.2. Global Offset Table Protection

Partial RELRO and hence lazy binding of DSOs is considered as default in the today's GNU linker implementation[5]. However, as presented in section 3.4, the *.got.plt* part of the Global Offset Table is vulnerable (Figure 3). To retrospectively utilize full RELRO, the entire GOT should be marked as read-only. To achieve this, first, the lazy binding mechanism must be deactivated, so that all relocations are performed at load-time. Considering that memory permissions apply on a per page basis, it must be assured that the *.got.plt* section is located in a dedicated part of a segment that may be marked as read-only without compromising the remaining *.data* segment. Finally, to complete the protection of the GOT, the function *_init()* needs to be modified so that it becomes responsible for altering permissions of the dedicated segment to read-only right after it has been populated and before the function *main()* has been initiated. Since all DSOs are resolved at load-time, this technique sacrifices startup time of programs in order to make sure that GOT tampering attacks are entirely eliminated.

5. Implementation

BinProtect presents a tool capable of retrospectively hardening security of programs in form of binary objects. Unprotected binaries should obtain protection similar to

5 The GNU linker: http://unixhelp.ed.ac.uk/CGI/man-cgi?ld, online Jan 2015.

mechanisms provided by modern compilers, as described in section 3. This section outlines implementation details of the techniques that are being shown in section 4.

5.1 Execution Prevention: NX-Bit

By default, the Linux kernel allocates a stack segment with execution permissions, in case the target ELF file misses a *PT_GNU_STACK* program header. To prevent execution on the stack segment, BinProtect first checks the presence of this header and makes sure that it does not contain the execution permission. In case the ELF file lacks the *PT_GNU_STACK* program header, BinProtect incorporates it into the ELF file to achieve the desired behavior.

5.2 Stack Protection

BinProtect transforms binaries in a way that they become able to check integrity of their activation records by means of a *shadow stack* mechanism that is incorporated into binaries. A shadow stack can be understood as an abstract data structure representing a collection of data, managed in a Last-In-First-Out (LIFO) manner. It allocates space for sensitive data within a separate memory region, which is maintained by a dedicated shadow stack pointer. The shadow stack mechanism maintains return addresses of currently active functions. After modification, the hardened functions become able to protect their return address as part of the function's prologue. The return address is stored on the shadow stack before any space for local variables is allocated on the stack and any function related instruction is executed. Before function return, as part of the epilogue, the modified functions perform integrity checks by comparing their return address with its copy on the shadow stack.

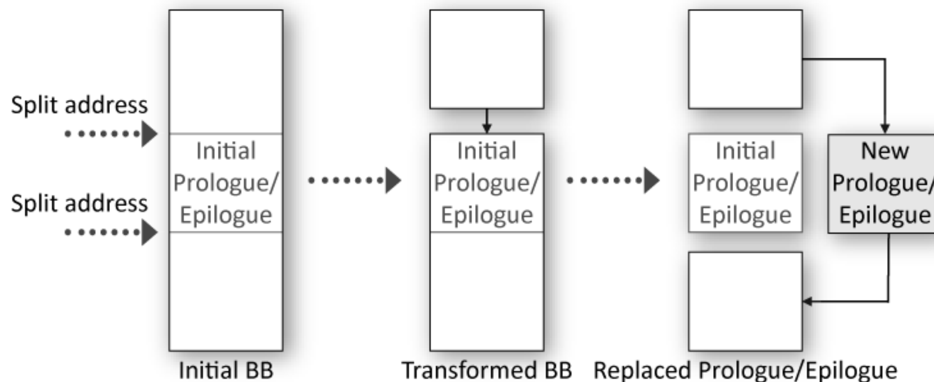

Figure 2: Basic block including initial prologue or epilogue information is transformed. Affected information is encapsulated within its own basic block, and subsequently exchanged with a new basic block.

Basic Block Transformation: BinProtect, in combination with PatchAPI, considers binaries as a flow of basic blocks interconnected by edges in form of a CFG. Since the size of basic blocks is determined by control flow properties of the code, the function's prologue and epilogue often present only a part of a basic block belonging to a particular function. To modify or replace a function's prologue or epilogue on the basis of control flow information, BinProtect first

localizes and encapsulates the affected part of the basic block. This is performed with help of information determined by common calling conventions. Calling conventions present a set of steps to be performed when calling and returning from functions. Thus, the tasks to be performed at function calls are clearly distributed among the caller and callee. Conventions to be performed by the callee, are usually performed within function's prologue and epilogue. With this information, BinProtect localizes the start and end of both prologue and epilogue within the associated basic block. After these locations have been determined, BinProtect transforms the basic blocks to enable code injection (Figure 2).

Code Injection: After the CFG transformations, presented within the previous step, the prologue or epilogue information has been extracted and encapsulated within its own basic block, as presented in Figure 2. The next step performed by BinProtect is the injection of raw binary code containing modified prologue or epilogue information. To achieve this, BinProtect creates a new basic block, inserts code into this basic block, and finally redirects edges previously directed to and from the initial prologue or epilogue to the created basic block. Figure 2 presents the steps performed by BinProtect to inject new prologue and epilogue information into the binary. The modified prologue becomes responsible for saving the function's return address within the shadow stack. Respectively, the modified epilogue contains instructions to perform integrity checks.

5.3 Standard C Library Consolidation

BinProtect intercepts and redirects calls to unsafe standard C library functions. Instead, equally named wrapper functions are executed. These are provided in form of a shared library. The wrapper functions are basically integrated as an intermediate layer between the binary itself and libc. Therefore, our prototype first determines whether libc has been *statically* or *dynamically* linked against the binary in question. This is done by means of the static analysis capabilities of Dyninst. In case libc has been *statically linked* against the target application, our prototype injects safe wrapper implementations directly into the binary. For this, Dyninst extracts wrapper raw binary code information from the provided shared object file and subsequently injects the gathered information into the ELF file to be modified. Dyninst prepares a new section, containing the injected code and assigns it to a new code segment of the target ELF file. Thus, the new code segment may be loaded into memory at program start. To allow the wrappers to call functions of further libraries, Dyninst integrates additional library dependencies into the ELF file and appropriately modifies the sections required for dynamic relocation. Finally, to redirect calls towards integrated wrappers, trampolines are injected in place of the original function's first instruction.

In case the libc has been *dynamically linked* against the target ELF file, our prototype inserts a dependency to the shared library containing safe wrapper implementations. To make the dynamic linker populate the GOT with addresses to our wrapper functions, BinProtect makes the dynamic linker load the provided library before libc. The ELF format determines the order of the dynamically

linked libraries to be loaded within the section *.dynamic*. This means, in order to intercept specific functions of libc, BinProtect swaps the library dependency entries in favor of our implementation. In addition, to allow the intercepted library functions call the original functions, BinProtect statically rewrites the function *_init()* of the original binary in a way that it resolves the affected symbol names and stores the associated libc function addresses within static variables before execution of the function *main()* is initiated.

5.4 Global Offset Table Protection

The part of the *.data* segment, which maintains addresses of functions and DSOs, may be potentially abused for malicious purposes if not marked as read-only. Partial RELRO reduces the attack vector by making the dynamic linker mark the affected part of the *.data* segment as read-only after it has completed relocations at load-time. The exact position and size of the affected memory region is stated within the ELF program header *PT_GNU_RELRO*. As discussed in section 3.1, the NX-bit protects execution on a per page basis. This implies that the linker is able to safely mark relocations as read-only only if they precede the page boundary to the *.data* section. Thus, in case partial RELRO has been activated, the linker arranges the *.data* segment (which comprises among others the sections *.init_array*, *.fini_array*, *.got*, *.got.plt*, *.data*, *.bss*, etc.) in such a way that the section *.got* is aligned with the end of a page that is positioned adjacently to the page containing the *.data* section. This is shown in Figure 3. Hence, the memory region containing relocations may be safely marked as read-only, whereas permissions of the section *.got.plt,* being located in the same page as the *.data* section, may not be changed as the adjacent *.data* section is part of the same page. To make the program benefit from security advantages provided by the full RELRO mechanism, BinProtect performs the following steps:

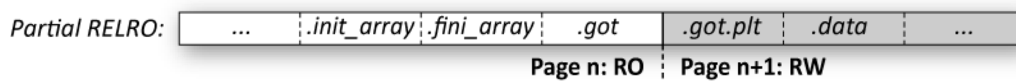

Figure 3: For partial RELRO, the linker distributes sections across different pages so that the .got.plt must remain writable after dynamic relocations have been performed.

Eager binding: The section *.dynamic* maintains information required during the process of dynamic linking. To make the dynamic linker perform all relocations at load-time, BinProtect integrates entries of type *DT_BIND_NOW* and *DT_FLAGS_1* into the *.dynamic* section of the original ELF file.

Relocation of .got.plt: To safely mark the entire GOT as read-only without affecting the remaining *.data* segment, BinProtect relocates the section *.got.plt* to a dedicated segment that is incorporated into the ELF file. This is done by specifying the virtual address of the *.got.plt* section to be within the range of the integrated segment. This address needs to be passed to the dynamic linker by adopting the *.dynamic* section entry of the type *DT_PLTGOT*. This way, the dynamic linker can find the relocated *.got.plt* section. In addition, to make sure that symbols of shared library objects are correctly relocated, offsets of entries

inside *.rel.plt* need to be adopted. Finally, to retain original functionality of the application, our prototype redirects pointers of the PLT to the relocated *.got.plt*.

Relocation read-only: The transformed application needs to be able to mark the entries inside *.got.plt* as read-only. To achieve this, BinProtect extends the function *_init()* in the same way as presented in section 5.3. The function *_init()* utilizes *mprotect()* to change permissions of the dedicated segment comprising the *.got.plt* at run-time and hence completes the protection of the GOT.

6. Evaluation

We evaluated BinProtect by effectively eliminating the effects of a security bug, which we have implemented for testing purposes into a local version of the *GNU tar* archiving utility. We demonstrate the potential of BinProtect by showing how BinProtect mitigates the security impacts of the bug. Also, to make concise deductions about the additional overhead induced by BinProtect, we assess measurements of the code size and execution time of three retrospectively hardened computational intensive applications.

The security bug exposes the unsafe libc function *strcpy()*, which allows to spill over bounds of buffers. Consequently, the attacker is able to inject and execute malicious code on the stack or heap by adapting the function's control flow structures. In addition, the bug enables a direct manipulation of GOT entries. The protection mechanisms of BinProtect successfully apply on different levels: The NX bit prevents execution on the stack and heap. Full RELRO hinders the attacker from modifying GOT entries. The stack protection detects return address corruptions. And finally, BinProtect intercepts the function *strcpy()* and prevents modifications of the function's control flow in the first place. As a result, BinProtect completely eliminates the security impacts of the bug.

Binary	***Hardened functions***	***Size increase***	***Time increase***
GNU tar, v1.27.1	807 (of 807)	38.7 %	4,7 %
Gzip, v1.6	67 (of 67)	86.3 %	2.3 %
Bzip2, v1.0.6	36 (of 36)	137.5 %	3.5 %

Table 1: Size and execution time increase after applications have been processed by BinProtect. Measurements have been taken of presented applications compressing 500 MB of data.

BinProtect adds overhead concerning code size and execution time to the original binaries. Table 1 illustrates the number of hardened functions and the resulted increase in code size and average execution time of three archiving utilities. Therefore, we collected 20 time measurements of each application by compressing data of 500 MB in size. The stated increase in size is the result of the performed binary transformations: To enable object code modifications of variable size, Dyninst extracts the targeted parts of the code and copies them to a separate location in the binary. Trampoline-based code constructs are inserted into the original code regions in order to access the applied modifications. Thus, the original code remains in the binary and hence blows up its size. This may present an issue for systems with limited resources. The increase in execution time, however, may be neglected within the context of general purpose applications.

7. Outlook and Conclusion

We have presented BinProtect, a static binary hardening tool, capable of mitigating potential buffer overflow vulnerabilities. This is achieved by means of cutting-edge static analysis and binary rewriting capabilities of Dyninst and PatchAPI. BinProtect performs instruction level binary modification based on function and control flow information, without the need for source code or debugging information. Applied binary transformations preserve structural validity of resulting binaries by utilizing PatchAPI's CFG transformation algebra. In addition, BinProtect directly modifies the ELF structure to utilize run-time security mechanisms that are enforced by both, the dynamic linker and Linux kernel. Concluding, our evaluation shows effective protection against a security bug, inserted into a copy of the tar archiving tool, with negligible time overhead.

References

[1] A. One, "Smashing the stack for fun and profit," *Phrack magazine,* vol. 49, no. 14, 1996.

[2] H. Orman, "The Morris worm: a fifteen-year perspective," in *IEEE Security & Privacy*, 2003.

[3] P. Wagle and C. Cowan, "Stackguard: Simple stack smash protection for gcc," in *Proceedings of the GCC Developers Summit*, Citeseer, 2003, pp. 243-255.

[4] Stack Shield, "A stack smashing technique protection tool for Linux," 2011. [Online]. Available: http://www.angelfire.com/sk/stackshield/. [Accessed Jan 2015].

[5] A. van de Ven, "New security enhancements in Red Hat Enterprise Linux v. 3, update 3," *Raleigh, North Carolina, USA: Red Hat,* 2004.

[6] A. Baratloo, N. Singh, and T. K. Tsai, "Transparent Run-Time Defense Against Stack-Smashing Attacks," *USENIX Annual Technical Conference, General Track,* pp. 251--262, 2000.

[7] TIS Comittee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2," *TIS Committee,* 1995.

[8] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *International Journal of High Performance Computing Applications,* vol. 14, no. 4, pp. 317-329, 2000.

[9] A. R. Bernat and B. P. Miller, "Structured binary editing with a cfg transformation algebra," *Reverse Engineering (WCRE), 2012 19th Working Conference on,* pp. 9--18, 2012.

[10] M. Prasad and T. C. Chiueh, "A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks," *USENIX Annual Technical Conference, General Track,* pp. 211--224, 2003.

[11] C. Zhang, L. Duan, T. Wei, and W. Zou, "SecGOT: Secure Global Offset Tables in ELF Executables," *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering,* 2013.

[12] R. Jones and P. Kelly, "Documentation for the proect: bounds checking for C," [Online]. Available: http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html. [Accessed Jan 2015].

[13] Levine, John R., Linkers & Loaders, San Francisco: Morgan Kaufmann Publishers, 1999.

[14] Styx^, "Infecting loadable kernel modules," *Phrack magazine,* vol. 68, no. 11, 2012.

[15] S. Cesare, "Shared library call redirection via ELF PLT infection," *Phrack magazine,* vol. 56, no. 7, 2000.

[16] S. Designer, "return-to-libc attack," *Bugtraq, Aug,* 1997.