Protecting global and static variables from buffer overflow attacks

Yves Younan Frank Piessens Wouter Joosen
DistriNet, Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200a, B3001 Leuven, Belgium
E-mail: {yvesy,frank,wouter}@cs.kuleuven.ac.be

Abstract

Many countermeasures exist to protect the stack and heap from code injection attacks, however very few countermeasures exist that will specifically protect global and static variables from attack. In this paper we suggest a way of protecting global and static variables from these type of attacks, with negligible performance and memory overheads. Our approach is based on the idea of separating data in de data segment based on its type. These separated areas are then protected from each other by a guard page. This prevents a buffer overflow from overwriting data or code pointers, in turn preventing attackers from being able to perform a code injection attack.

1 Introduction

Vulnerabilities that could lead to code injection attacks are a significant threat to the security of a system. The most common form of code injection attack is the stack based buffer overflow and many countermeasures [39] exist that protect against this attack. The heap is also a source of buffer overflows and there are some countermeasures that will protect the heap from attack, however very few countermeasures currently exist that will protect global and static variables from these types of attack.

In [40] we describe a global approach to protect against code injection attacks by separating data that the operating system relies on from regular user data. This technique has proven succesful from a security perspective as well as from a performance perspective in protecting against heap-based [42] and stack-based [41] overflows. While the basic idea of separating user data from system data is the same in these two countermeasures, the actual approaches that need to be taken to separate the data turn out to be quite different. In this paper we describe the details of how to apply this separation idea to protect against attacks on global or static variables. Combining these three countermeasures leads to a strong overflow protection against dynamically, automatically and statically allocated memory. A major advantage of these countermeasures is that they can be applied automatically without programmer intervention, they are automatically added when compiling and linking with these countermeasures.

The paper is structured as follows: Section 2 describes how a buffer overflow in this region can be used by an attacker to gain control of the execution flow. Section 3 describes the main design principles of our countermeasure, while Section 4 discusses limitations and how we plan to implement the countermeasure. Section 5 compares our approach to other approaches. Finally, section 6 contains our conclusion.

2 Static and global variables

In this section we describe how the memory that stores static and global variables is organized and then examine how an attacker could use a buffer overflow on one of this variables to gain control of the execution flow on the IA32 architecture [15].

2.1 Memory layout

Figure 1 depicts the memory layout of a Linux process on the IA32 architecture. Code is stored in the text segment, while local (automatic) variables are stored on the stack. Global, static data and the heap (dynamic memory) are all stored in the data segment. The data segment however also contains other important information that the operating system relies on to execute the program.

Figure 2 provides an overview of the layout of the data segment of a typical program. Static and global variables which have been initialized at compile time are stored in the data section, followed by the section containing the exception handling frame, which holds information needed to handle exceptions in languages that support them (like

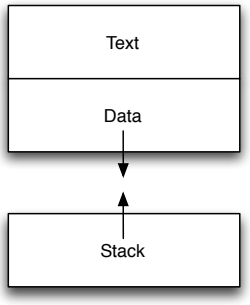

Figure 1. Basic memory layout on Linux/IA32

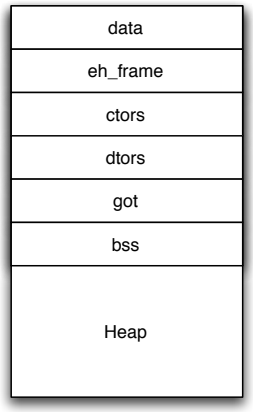

Figure 2. Layout of the data segment

C++). This section is followed by the ctors and dtors sections, these execute registered functions at respectively program start and program finish. Next, is the global offset table, which is used by position independent code[1] to address absolute memory locations. Its values are set by the runtime linker when new code is loaded, it also holds absolute memory addresses for library functions. Static and global variables which have not been initialized are stored in the bss section, they are initialized to 0 in this area.

2.2 Exploitation

If attackers can overflow a variable in the data section, they could easily overwrite data stored in the other sections. Two favorite targets of attackers are the dtors and the got sections.

The dtors section is comprised of a list of pointers to functions to execute when the program terminates, terminated with a NULL. If an attacker can overwrite these pointers, his code will be executed when the program terminates [30].

The global offset table contains the absolute address of shared library functions which are used by the procedure linkage table to execute functions which have to be loaded from a shared library at runtime. If an attacker modifies the address of one of these functions (e.g. the *printf* function) to point to injected code, the program will execute that code when the library function is supposed to be called.

Overflows in the bss section can not overwrite any of the other sections because the bss section is stored last. However, immediately following the bss section is the heap, thus an attacker could use an overflow in the bss section to perform a heap-based buffer overflow [33] which could also allow him to gain control of the execution flow.

3 Countermeasure

In this section we propose a countermeasure which could protect against these types of attack. By separating the data which can be used by an attacker to perform a code injection attack from data which could modify control flow if changed, we can protect against this type of attack. The concept of this countermeasure is straightforward: by reorganizing the data segment and making sure that all important information comes before any arrays, we can prevent most attacks. We make sure that these arrays can not write into the heap by placing a guard page[2] between the last ar-

[1] PIC is code that can be loaded at any address, it does not address any absolute memory locations

[2] A guard page is page of memory where no permission to read or to write has been set. Any access to such a page will cause the program to terminate.

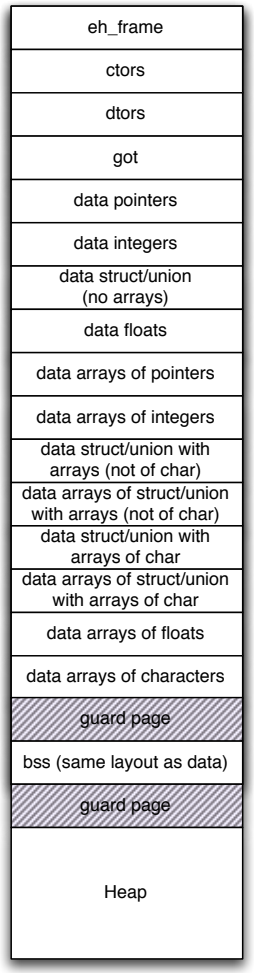

Figure 3. Modified layout of the data segment

ray and the heap[3]. Since the size of the data segment is known at compile time, adding such a guard page at load time does not introduce any problems.

Figure 3 illustrates the modified memory layout: first is the memory which only contains data used by the loader: the constructors, destructors, global offset table, the exception handling frame, etc. Since a program should not be able to change them, these can be stored at the start of the segment. While a major avenue for attack has been closed off by preventing arrays from overflowing into the destructors section and the global offset table, an attacker could still use an overflow in the data or bss section to overwrite data in that section. To prevent this from occurring we divide the data into three categories: not-overflowable (ordered by the risk they pose if they are a target for attack), overflowable (ordered from less likely to overflow to more likely) but also a target of attacks and just overflowable (ordered by how likely they will overflow).

Pointers could be overwritten to perform a code injection attack, but are not exploitable on their own, so we place them in the first category. Integers can hold pointers or can be used as offsets to a pointer, so they could also be used to perform indirect pointer overwrites. Although they are overflowable, they will never use more than the memory allocated for them since they will wrap around zero on overflow, as such we can place them in the same category as pointers. Structures and unions that do not contain any arrays come next, they could contain pointers that could be overwritten, but do not contain any arrays so they are not overflowable. The last element of the first category are floats, they are not overflowable and are also not a likely target for attack.

The second category contains data which can overflow in theory, but which could also be used to perform an indirect pointer overwrite. This data is sorted by the risk posed if it is attacked: higher risk data is stored first so that if an overflow occurs, it can only overwrite equally or less risky data. The first element in this category are arrays of pointer, followed by arrays of integers. Next we place structures and unions which contain arrays, but not arrays of characters, arrays of these structures and unions, structures and unions which contain arrays of characters and finally arrays of these structures.

The third category only contains data which is overflowable, ordered by how likely they are to be overflown. The first elements stored here are arrays of floats, these do not contain information which could easily lead to a code injection attack and could overflow. Arrays of characters, which are most often targeted by attackers, because they are often used with vulnerable string manipulation functions (e.g. *strcpy*), are placed last.

The data layout that these three categories provide are used for both the data and bss sections which are stored next to each other. To protect the bss section from the arrays of characters from the data section, we place a guard page between the two sections.

By separating the data which can influence control from data which is can be modified by the user, we can protect against buffer overflow attacks in these sections.

4 Discussion

It may still be possible to perform a code injection attack using a buffer overflow on a structure that contains both an array of characters and a pointer since these types of structures will be stored in a contiguous region of memory. This is a limitation of the approach that can not easily

[3] A page on IA32 is 4096 bytes, this means that if we want to add a guard page, the memory preceding the guard page will need to be aligned on 4096 bytes. Adding such a page after all memory locations would be introduce a prohibitive memory overhead [29]

be fixed because the C standard[18] mandates that all structures must be stored in order in contiguous memory. Many other countermeasures, including bounds checkers suffer from this limitation. Given that correct code can rely on this feature of the C language, it is hard to protect against such an attack. To reduce the risk of attack, we treat structures containing arrays of characters different from other structures: by storing them below regular structures. While this does not solve the problem, it reduces the risk of successful exploitation because attackers can only overflow information stored below this type of structure.

Integer errors could also be used to perform a buffer overflow because they could be used to overwrite arbitrary memory locations, without requiring a contiguous buffer overflow. This would bypass the protection provided by the guard page. This is an important limitation in our approach and must be taken into account when applying this countermeasure.

Implementation of this countermeasure will require some substantial modifications to the compiler, the linker and the loader, however because only the layout is changed, it should not bring any extra performance overhead with it. Because the size of all the sections are known at load time, changing the layout will not add much memory overhead either: each section followed by a guard page will have to be aligned to page size, which means that the maximum overhead per section would be 4095 bytes (if only 1 byte is used on the last page of that section). Since only 2 guard pages are used, the countermeasure has a maximum overhead of 8190 bytes. The guard pages themselves will only take up virtual memory, since they are not accessed, no physical memory is consumed.

5 Related work

Not many countermeasures exist that specifically try to protect this type of data. Although some of the more global approaches (like bounds checking) will also protect global and static variables. In this section we will first discuss the only other countermeasure that specifically targets this memory and will then briefly discuss the more global approaches.

Drepper [11] implemented a countermeasure which reorganizes the data segment so that the data and bss sections are placed similarly to the way we describe earlier. However he does not reorganize the data within these sections so arrays of characters could still overwrite pointers in these sections. He also does not add guard pages which will protect the heap from being attacked by the bss section.

5.1 Alternative approaches

Many alternative approaches exist that try and protect against buffer overflow attacks. In this section we will briefly discuss the most important types of countermeasures. A more extensive discussion can be found in [38, 12].

5.1.1 Probabilistic countermeasures

Many countermeasures make use of randomness when protecting against attacks. Many different approaches exist when using randomness for protection. Canary-based countermeasures [9, 13, 23, 31] use a secret random number that is stored before an important memory location: if the random number has changed after some operations have been performed, then an attack has been detected. Memory-obfuscation countermeasures [8, 5] encrypt (usually with XOR) important memory locations or other information using random numbers. Memory layout randomizers [35, 4, 36, 6] randomize the layout of memory: by loading the stack and heap at random addresses and by placing random gaps between objects. Instruction set randomizers [3, 19] encrypt the instructions while in memory and will decrypt them before execution.

While these approaches are often efficient, they rely on keeping memory locations secret. However, programs that contain buffer overflows could also contain ”buffer overreads” (e.g. a string which is copied via *strncpy* but not explicitly null-terminated could leak information) or other vulnerabilities like format string vulnerabilities, which allow attackers to print out memory locations. Such memory leaking vulnerabilities could allow attackers to bypass this type of countermeasure.

5.1.2 Bounds checkers

Bounds checking [20, 34, 2, 17, 25, 27, 32, 28] is a better solution to buffer overflows, however when implemented for C, it often has a severe impact on performance or may cause existing code to become incompatible with bounds checked code.

5.1.3 Safe languages

Safe languages are languages where it is generally not possible for any known code injection vulnerability to exist as the language constructs prevent them from occurring. A number of safe languages are available that will prevent these kinds of implementation vulnerabilities entirely. There are safe languages [16, 14, 26, 24, 10, 22, 37] that remain as close to C or C++ as possible, these are generally referred to as safe dialects of C. While some safe languages [7, 37] try to stay more compatible with existing C

programs, use of these languages may not always be practical for existing applications.

5.1.4 Execution monitors

In this section we describe two countermeasures that monitor the execution of a program and prevent transferring control-flow which could be unsafe.

Program shepherding [21] is a technique that monitors the execution of a program and will disallow control-flow transfers[4] that are not considered safe. An example of a use for shepherding is to enforce return instructions to only return to the instruction after the call site. The proposed implementation of this countermeasure is done using a runtime binary interpreter. As a result, the performance impact of this countermeasure is significant for some programs, but acceptable for others.

Control-flow integrity [1] determines a program's control flow graph beforehand and ensures that the program adheres to it. It does this by assigning a unique ID to each possible control flow destination of a control flow transfer. Before transferring control flow to such a destination, the ID of the destination is compared to the expected ID, and if they are equal, the program proceeds as normal. This approach, while strong and in the same efficiency range as our approach, does not protect against non-control data attacks.

6 Conclusion

Many countermeasures exist to protect against attacks on stack-based buffer overflows, however only very few exist that will effectively protect against attacks on global and static variables with a low performance overhead. In this paper we suggested an approach which would better protect this region of memory from attack while only having negligible performance and memory overhead.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, Alexandria, Virginia, U.S.A., Nov. 2005. ACM.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Florida, U.S.A., June 1994. ACM.

[3] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 281–289, Washington, District of Columbia, U.S.A., Oct. 2003. ACM.

[4] 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*, pages 105–120, Washington, District of Columbia, U.S.A., Aug. 2003. USENIX Association.

[5] S. Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, Paris, France, July 2008. Springer.

[6] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, Baltimore, MD, August 2005. USENIX Association.

[7] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244, San Diego, California, U.S.A., 2003. ACM.

[8] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, District of Columbia, U.S.A., Aug. 2003. USENIX Association.

[9] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, U.S.A., Jan. 1998. USENIX Association.

[10] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, pages 69–80, San Diego, California, U.S.A., June 2003. ACM.

[11] U. Drepper. Security enhancements in redhat enterprise linux (beside selinux). `http://people.redhat.com/drepper/nonselsec.pdf`, Dec. 2005.

[12] U. Erlingsson. Low-level software security: Attacks and defenses. Technical Report MSR-TR-2007-153, Microsoft Research, Nov. 2007.

[13] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Technical report, IBM Research Divison, Tokyo Research Laboratory, June 2000.

[14] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

[15] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 2001. Order Nr 245470.

[16] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual*

[4] Such a control flow transfer occurs when e.g., a *call* or *ret* instruction is executed.

Technical Conference, pages 275–288, Monterey, California, U.S.A., June 2002. USENIX Association.

[17] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, number 009-02 in Linköping Electronic Articles in Computer and Information Science, pages 13–26, Linköping, Sweden, 1997. Linköping University Electronic Press.

[18] JTC 1/SC 22/WG 14. ISO/IEC 9899:1999: Programming languages – C. Technical report, International Organization for Standards, 1999.

[19] 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 (CCS2003)*, pages 272–280, Washington, District of Columbia, U.S.A., Oct. 2003. ACM.

[20] S. C. Kendall. Bcc: Runtime checking for C programs. In *Proceedings of the USENIX Summer 1983 Conference*, pages 5–16, Toronto, Ontario, Canada, July 1983. USENIX Association.

[21] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, U.S.A., Aug. 2002. USENIX Association.

[22] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the International Conference on Compilers Architecture and Synthesis for Embedded Systems*, pages 288–297, Grenoble, France, Oct. 2002.

[23] A. Krennmair. ContraPolice: a libc extension for protecting applications from heap-smashing attacks. `http://www.synflood.at/contrapolice/`, Nov. 2003.

[24] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fähndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May/June 2004.

[25] K.-S. Lhee and S. J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–90, San Francisco, California, U.S.A., Aug. 2002. USENIX Association.

[26] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, Oregon, U.S.A., Jan. 2002. ACM.

[27] Y. Oiwa, T. Sekiguchi, E. Sumii, and A. Yonezawa. Fail-safe ANSI-C compiler: An approach to making C programs secure: Progress report. In *Proceedings of International Symposium on Software Security 2002*, pages 133–153, Tokyo, Japan, Nov. 2002.

[28] H. Patil and C. N. Fischer. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practice and Experience*, 27(1):87–110, January 1997.

[29] B. Perens. Electric fence 2.0.5. `http://perens.com/FreeSoftware/`.

[30] J. M. B. Rivas. Overwriting the .dtors section. Posted on the Bugtraq mailinglist `http://www.securityfocus.com/archive/1/150396`, Dec. 2000.

[31] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Runtime detection of heap-based overflows. In *Proceedings of the 17th Large Installation Systems Administrators Conference*, pages 51–60, San Diego, California, U.S.A., Oct. 2003. USENIX Association.

[32] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, California, U.S.A., Feb. 2004. Internet Society.

[33] Solar Designer. JPEG COM marker processing vulnerability in netscape browsers. `http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt`, July 2000.

[34] J. L. Steffen. Adding run-time checking to the portable C compiler. *Software: Practice and Experience*, 22(4):305–316, Apr. 1992. ISSN: 0038-0644.

[35] The PaX Team. Documentation for the PaX project. `http://pageexec.virtualave.net/docs/`.

[36] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269, Florence, Italy, Oct. 2003. IEEE Computer Society, IEEE Press.

[37] W. Xu, D. C. DuVarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 117–126, Newport Beach, California, U.S.A., October-November 2004. ACM, ACM Press.

[38] Y. Younan. *Efficient Countermeasures for Software Vulnerabilities due to Memory Management Errors*. PhD thesis, Katholieke Universiteit Leuven, 2008.

[39] Y. Younan, W. Joosen, and F. Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.

[40] Y. Younan, W. Joosen, and F. Piessens. A methodology for designing countermeasures against current and future code injection attacks. In *Proceedings of the Third IEEE International Information Assurance Workshop 2005 (IWIA2005)*, College Park, Maryland, U.S.A., Mar. 2005. IEEE, IEEE Press.

[41] Y. Younan, W. Joosen, and F. Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proceedings of the International Conference on Information and Communication Security (ICICS 2006)*, Raleigh, North Carolina, U.S.A., Dec. 2006.

[42] Y. Younan, D. Pozza, F. Piessens, and W. Joosen. Extended protection against stack smashing attacks without performance loss. In *Proceedings of the Twenty-Second Annual Computer Security Applications Conference*, Miami, Florida, U.S.A., Dec. 2006. IEEE Press.