Protecting global and static variables from buffer overflow attacks without overhead

Yves Younan
Frank Piessens
Wouter Joosen
Report CW 463, October 2006

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Protecting global and static variables from buffer overflow attacks without overhead

Yves Younan
Frank Piessens
Wouter Joosen
Report CW 463, October 2006

Department of Computer Science, K.U.Leuven

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.

CR Subject Classification : K6.5, D3.4, D4.2

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 [22] 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 [23] 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 [24] and stack-based [25] overflows. While the basic idea of separating user data from system data is the same in these two countermeasures, their actual implementations turn out to be quite different. In this paper we describe the details of how to apply this separation to protect against attacks on global or static variables. Combining the three countermeasures leads to a strong overflow protection against dynamically, automatically and statically allocated memory.

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.

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 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 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. It's values are set

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

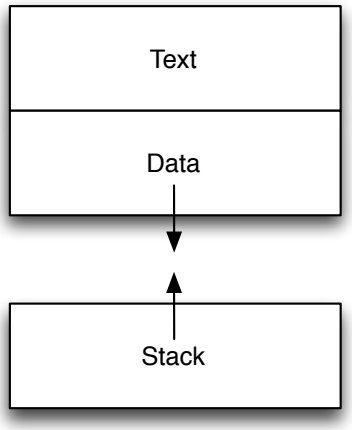

Figure 1: Memory layout

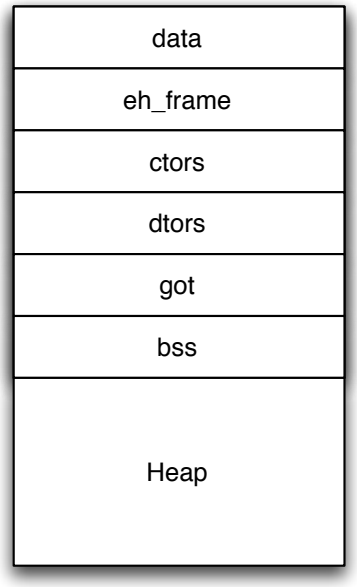

Figure 2: Layout of the data segment

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 [14].

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 [17] 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 array and the heap. 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 (like pointers) in its own 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),

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.

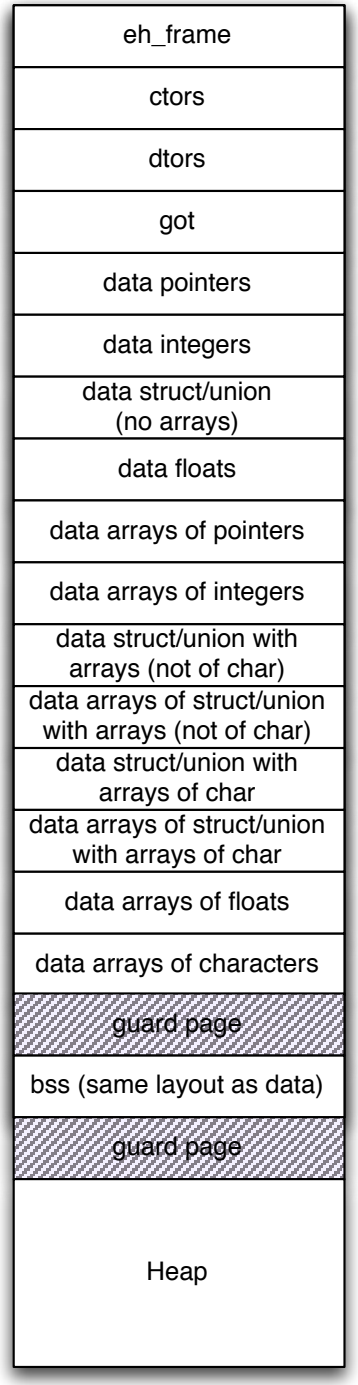

Figure 3: Modified layout of the data segment

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 don't 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 be fixed because the C standard mandates that all structures must be stored in order in contiguous memory. However this type of vulnerability does not occur often in practice so the limitation is unlikely to significantly undermine the protection.

Integer errors could also be used to perform a buffer overflow (e.g. if they are used as an offset for a write to an array). However, because they could be used to overwrite arbitrary memory locations, they could bypass the protection provided by the guard page.

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 won't add much memory overhead either.

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 [5] 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.

Bounds checking [2, 8–10, 12, 13, 15, 18, 21] is the perfect 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.

Safe languages eliminate the vulnerabilities that can result in code injection attacks by constructing a language that will prevent the vulnerabilities from occurring by combining: restrictions on pointer use, statically determining safety of code and adding runtime checking for code which can't be declared safe statically. Although some languages try to remain as close to C [6, 7, 11] as possible (called safe dialects of C), it may not always be practical to use them for existing applications.

Address randomization [3, 4, 19, 20] is a technique that attempts to provide security by modifying the locations of objects in memory for different runs of a program, however the randomization is limited in 32-bit systems (usually to 16 bits for the heap) and as a result may be inadequate for a determined attacker [16].

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 each possible control flow destination of a control flow transfer, a unique ID. 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. Control-flow integrity relies on important assumptions, which may not always be available: no data section may be executable and no code section may be writable. Performance overhead may be acceptable for some application but may be prohibitive for others.

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] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, Alexandria, Virginia, U.S.A., November 2005. ACM.

[2] Todd M. Austin, Scott E. Breach, and Gurindar 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] Sandeep Bhatkar, Daniel 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., August 2003. USENIX Association.

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

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

[6] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James 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.

[7] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, California, U.S.A., June 2002. USENIX Association.

[8] Richard W. M. Jones and Paul 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.

[9] Samuel 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.

[10] Kyung-Suk Lhee and Steve 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., August 2002. USENIX Association.

[11] George Necula, Scott McPeak, and Westley 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., January 2002. ACM.

[12] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori 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, November 2002.

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

[14] Juan M. Bello Rivas. Overwriting the .dtors section. Posted on the Bugtraq mailinglist `http://www.securityfocus.com/archive/1/150396`, December 2000.

[15] Olatunji Ruwase and Monica 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., February 2004. Internet Society.

[16] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, Washington, District of Columbia, U.S.A., October 2004. ACM, ACM Press.

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

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

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

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

[21] Wei Xu, Daniel 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.

[22] Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.

[23] Yves Younan, Wouter Joosen, and Frank 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., March 2005. IEEE, IEEE Press.

[24] Yves Younan, Wouter Joosen, Frank Piessens, and Hans Van den Eynden. Security of memory allocators for C and C++. Technical Report CW419, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2005.

[25] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended protection against stack smashing attacks without performance loss. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, Miami Beach, Florida, U.S.A., December 2006. Accepted.