When stack protection does not protect the stack?

Pavel Dovgalyuk <pavel.dovgaluk@ispras.ru>
Vladimir Makarov <vladimir.makarov@novsu.ru>
Novgorod State University,
41, Bolshaya Sankt Peterburgskaya, Velikiy Novgorod, 173003, Russia

Abstract. The majority of software vulnerabilities originate from buffer overflow. Techniques to eliminate buffer overflows and limit their damage include secure programming, source code audit, binary code audit, static and dynamic code generation features. Modern compilers implement compile-time and execution time protection schemes, that include variables reordering, inserting canary value, and separate stack for return addresses. Our research is targeted to finding the breaches in the compiler protection methods. We tested MSVC, gcc, and clang and found that two of these compilers have flaws that allow exploiting buffer overwrite under certain conditions.

Keywords: buffer overflow; canary protection; gcc; msvc; clang

DOI: 10.15514/ISPRAS-2016-28(5)-3

For citation: P.M. Dovgalyuk, V.A. Makarov. When stack protection does not protect the stack? Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 55-72. DOI: 10.15514/ISPRAS-2016-28(5)-3

1. Introduction

Even though there is a lots of progress in mitigating attacks against buffer overflows and in building static analysis tools that attempt to detect these vulnerabilities, buffer overflows remain one of the top ranking vulnerabilities year over year [1].

Writing beyond the buffer boundaries results in an undesired modification of adjacent memory locations. This data corruption can be exploited by an attacker to change the control flow of the program [2].

Software flaws that allow buffer overflow are produced by the developers. There are two kinds of countermeasures against buffer overflows [2]. First group is targeted to finding a problem before software is deployed: secure programming, source code audit, binary code audit, automatic testing (including testing with static analysis tools).

There are various static analysis techniques that address buffer overflow problem [3], [4], [5]. Static techniques extract constraints from the code and try to find incorrect memory accesses. However, these techniques produce many false positives due to the challenges faced by static analysis, such as the limited precision of alias analysis and limited loop unrolling. They may also produce false negatives due to the limited ability of constraint solvers, that are used to recover the malicious inputs [6].

Another approach is reducing the damage caused by the overflow. It includes the attempts of stopping an application after overflow is detected and reducing bug exploitability chances (address space randomization, non-executable data segments).

Wilander and Kamkar describe in their paper [7] several code generation approaches for preventing buffer overflow exploitation: reordering local variables, inserting canary values, additional stack for return addresses, checking return addresses range, checking function pointers range, and wrapping unsafe library functions.

Variables reordering and canary values protecting the return address have low runtime overhead and greatly reduce the exploitability of the overflow bugs. Nowadays these techniques are adopted by the commodity compilers Microsoft Visual C++ (MSVC) and GNU C Compiler (gcc) [8], [9].

Attacker can try to bypass the protection using other kind of attack (format string, double free, integer overflow) or bruteforcing the canary value [10], [11], [12], [13].

After bypassing protection scheme an attacker can overwrite instruction pointer and therefore hijack the control flow. This kind of threat is well known and many prior researches focused on this problem [14].

Another possible threat is overwriting the stack frame pointer (SFP) [15]. SFP is often copied to the stack pointer (SP) in the epilog of the function. When SP gets incorrect value, then function returns to the address stored in stack pointed by such fake SP. There are some other possibilities for overwriting SP that result in reading incorrect return address from the ”stack“.

Overwriting stack and frame pointers may also alter function execution before exit, because it accesses local variables through these pointers. Attacker may substitute an address to overwrite variables in another memory region or output sensitive data from the program.

In our work we address stack and frame pointer attacks, related to the buffer overflow problem. Modern compilers try to reduce attacks damage. We decided to test the compilers and stated the main research question: can stack canaries protect from overwriting stack and frame pointers?

Finally, this work makes the following contributions:

- Survey of the prolog/epilog code generation techniques in modern compilers.

- Set of the snippets for generating different stack frame manipulation samples.
- Description of MSVC stack protection bug, which allows attacking callee-saved registers. Attacker can exploit this bug to pass by the stack protection.
- Description of MSVC and clang stack protection bugs, that allow overwriting stack pointer by calling vulnerable function.

2. Protection Against Stack Buffer Overflows Through the Code Generation

We investigated code generation techniques for local buffer overflow prevention. Traditional stack frame includes arguments of the function, return address, saved value of frame pointer register, and local variables [7]. On x86 stack grows from higher to lower addresses. Therefore local buffer overflow in a specific function may overwrite its return address (Figure 1).

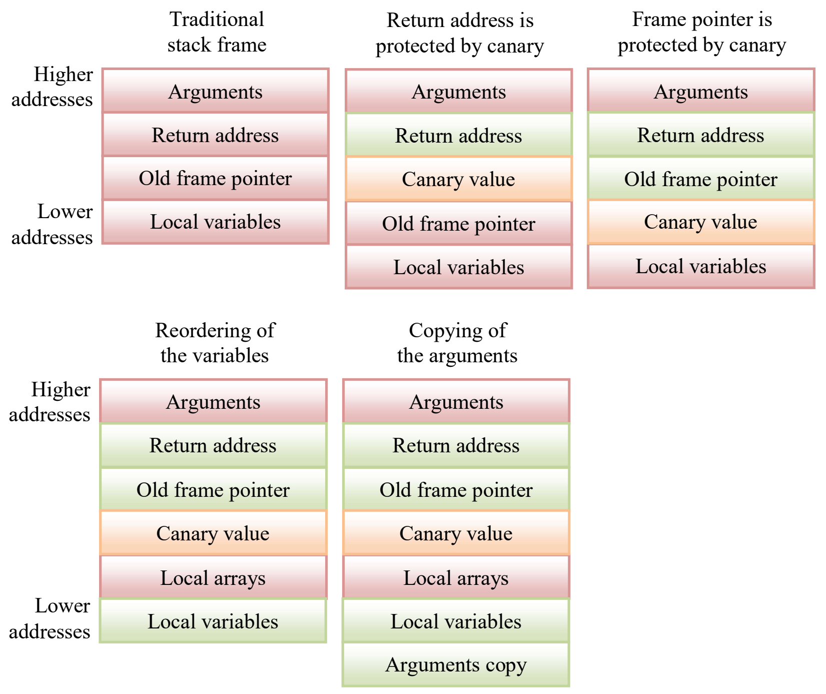

Fig. 1. Stack frame protection methods. Red frames are vulnerable. Green frames are safe.

There are multiple methods to protect from the control flow hijacking through the buffer overflow [7]. Modern compilers adopted embedding canary value into the stack frame, local variables reordering, and copying function arguments.

The structure of stack frame with these protection methods is shown in Figure 1.

2.1 Canary Value

Canary value is a local variable embedded into the stack frame to detect buffer overflows. When canary value is changed it means that some code accessed the stack frame through the wrong pointer. StackGuard concept invented by Crispin Cowan [16] is designed for using canary values to detect and stop stack-based buffer overflows targeting the return address.

Canary value is placed ”below“ the return address to detect when it is changed by user’s code. Overflow of the buffer, located in the lower addresses stack frame, targeted to overwrite the return address, will also overwrite canary value and the attack will be detected. Function checks the canary value before reading the return address from the stack. Program is terminated when canary value is incorrect, preventing buffer overflow exploitation.

Compiler developers enhanced this protection scheme. Canary value now protects not only the return address but also saved frame pointer. Prior protection schemes used predictable canary values. Attacker could ”overwrite“ them with the same values. Random canary values used in modern compilers provide efficient protection from return address and frame pointer overwriting [17], [18].

Canary values is not intended to protect other local variables when one of them overflows [17]. E.g., one local buffer may overwrite other local buffer in case of overflow. We do not address this problem in our research, because it requires more complex protection approach, which is not used in the compilers yet.

On the other hand, variables that are not addressed as buffers, may be protected by moving them to lower addresses in the stack. This protection is called variables reordering.

2.2 Reordering of the Variables

Compiler reorders variables to protect them from buffer overflows. Moving arrays to higher addresses protects other local variables from being overwritten. Buffer overflow can overwrite other variables only in the case when they are also buffers and located at higher addresses. Ordinary variables in single-buffer functions can feel safe.

Attacker can also overwrite arguments that are located at higher addresses, because belong to caller’s stack frame. This attack will be detected only at function’s exit. But values of the arguments are used before exit and attacker may affect function’s behavior before attack will be detected.

2.3 Copying Function Arguments

Compiler may protect function arguments from being overwritten by creating its copies. Arguments are protected by allocating extra space on the stack and copying their values below the local variables. The original argument values located after the return address are not used in the rest of the code [13], [17]. Therefore copying the arguments protects the function from using invalid values in the code between buffer overflow and function return.

3. Method of Analysis

To find flaws in protection mechanisms and answer the research question we decided to create multiple code snippets and compile them with different compilation options on modern compilers.

We focus on software compiled for 32-bit i386 platform. i386 family CPUs use esp register as stack pointer and in most cases use ebp as frame pointer. We tested two versions of MSVC — 2010 and 2015, two versions of gcc — 4.6.2 and 5.2.0, and clang 3.7.1.

All our snippets were on pure C, without C++-related features like exception handling. These snippets are targeted to generate different code structure of prolog and epilog parts of the functions. Prolog stores callee-saved registers and initializes the stack frame. Epilog restores the registers and exits from the function.

Prolog and epilog also include arguments copying, canary value initialization and verification embedded by the compiler.

3.1. Snippets Code Structure

We have found that the following features in program structure affect on prolog and epilog code generation.

- Function main. main function may align stack pointer on entry and restore it at exit, when it is required by the platform application binary interface.
- Variable length arrays (VLA). These arrays cannot be allocated in prolog, because its sizes are unknown. Therefore creating and overflowing them may affect the execution in unusual way.
- Aligned structures on the stack. When structures need an aligned address, compiler must align the stack pointer to allocate these structures. It changes the structure of prolog and epilog, because original unaligned stack pointer value must be restored at exit.
- `alloca` function is an alternative for variable-sized arrays. It allocates specified number of bytes on the stack. We included this function in our snippets, because MSVC doesn't support VLA.

Every snippet has at least one function which performs operations with stack variables. Function template includes the following operations:

- Stack buffer allocation. Buffers were allocated statically or dynamically depending on function variant.
- Reading data into local buffer with gets function. `gets` was selected to demonstrate whether stack frame may be overwritten with overflow or not.
- Output buffer with `printf` function.

We also tried adding different types of parameters and return values to the functions. These options didn't add any exploitable registers or memory cells in functions prologs and epilogs. Switching between calling conventions (cdecl, fastcall, stdcall) also didn't affect potential prolog/epilog vulnerabilities.

3.1. Compilation Options

We compiled our samples with different combinations of prolog/epilog code generation options. These options are supported by all tested compilers.

- Omit frame pointer. Omitting the frame pointer is the optimization when local variables are accessed directly though the stack pointer. If frame pointer is not used, then it cannot be hijacked by the attacker. But this option does not work in some examples, making frame pointer vulnerable to overwriting.
- Stack protection. Stack protection intended to guard return address and callee-saved registers from being overwritten. However, in some test cases frame pointer or saved registers were not protected and could be overwritten by the attacker.
- Code optimizations. Code in released software binaries is usually optimized by the compiler. We analyzed non-optimized code, but do not present its flaws here, because they probably will not appear in production code.

4. Prolog/epilog in Modern Compilers

Prolog and epilog for compiled templates include different parts. Some of them belong to user's code, others are inserted by the compiler for implicit operations like saving registers. Assembly code for generated functions included the following parts:

- Saving frame pointer register (`ebp`) and copying `esp` to `ebp`. This part is omitted if frame pointer is not used in this function.
- Pushing callee-saved registers into the stack.
- Aligning `esp` and saving its initial value into some register (e.g., `esi`).
- Allocating space for the local variables.
- Initializing the canary — prolog saves canary value as a hidden local variable.

- User’s code. This code can modify `esp` to allocate new variables or function parameters.
- Checking the canary.
- Restoring initial value of `esp`. `esp` may be restored by copying saved value from the register.
- Restoring values of the saved registers.
- Recovering `ebp`.
- Exitting the function. `ret` instruction loads return value stored in stack into the program counter.

The only mandatory action in this list is exitting from the function. Others depend on function structure and compiler options. The order of parts may change from compiler to compiler (e.g., the order of saving registers and initializing the canary may change).

We focus on overflows of the buffers located in the stack, but don’t consider the following attacks that could be performed with this overflow:

- Attack on local variables. As said before, we don’t focus on this attack, because protection mechanisms used by the compilers are not intended to stand against it.
- Direct overwrite of the return address. This attack is well known [14] and doesn’t require to be analyzed here. Stack canaries used by the compilers protect return address in the first place.
- Attack on non-optimized code. This code won’t appear in deployed software and therefore attackers do not interested in its vulnerabilities.

We detected several templates of prolog and epilog for MSVC, gcc, and clang compilers. Here we present an analysis of compiled samples, focusing on possible attack vectors.

4.1. Scheme of Protection

When compilers embed canary value into the stack, they insert initialization code into the prolog. This code puts some value into the canary variable. We found the following types of canaries in the generated samples.

Value from global variable. Canary value is read from some global variable. Local canary variable may be operated through `ebp` (or or through `esp`, if frame pointer is not used). Reading canary value through `ebp` protects this register from overwriting, because wrong register value will point to wrong canary value. This approach is used by gcc and clang.

Global variable `xor` frame pointer. Canary value is formed by `xor`’ing global variable with the value of frame pointer (`ebp`). When frame pointer is omitted, local variables are accessed through `esp`. In this case canary value includes `esp`, protecting it from corruption. This approach is used by MSVC.

4.2. MSVC 2010/2015

Microsoft visual C includes support of C99 subset. We tested two versions of MSVC — 2010 and 2015. We didn’t compile tests with variable-length arrays, because MSVC doesn’t support them.

MSVC supports stack frame protection with canary value (which is named ”security cookie“ in compiler documentation), local variables reordering, and function arguments copying.

Bray describes stack frame of MSVC 2003 and states that security cookie is disabled when `_alloca` function is used [8]. `_alloca` function is intended to allocate buffers in the stack frame dynamically. Therefore it can be used as a replacement of variable-length arrays.

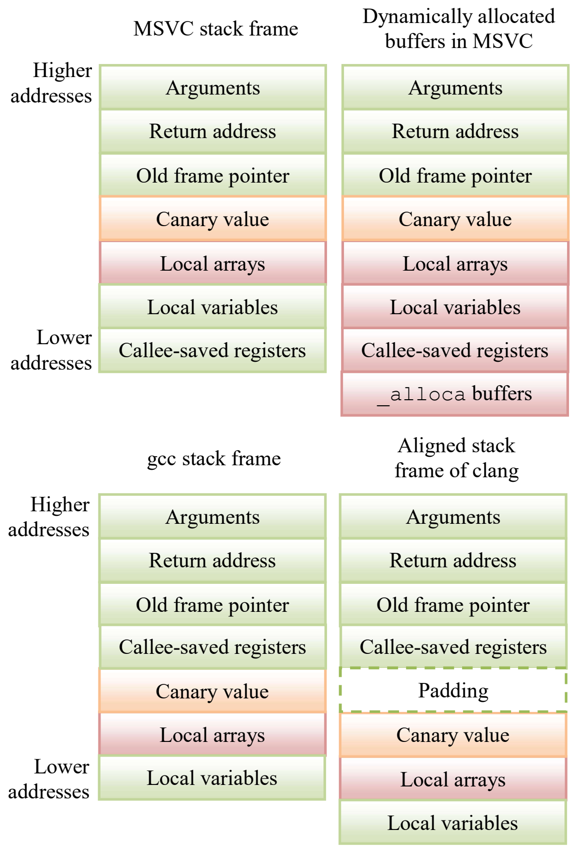

Fig. 2. Stack frame layouts used by the modern C compilers.

```
void func(void)
{
 int sz;
 char *buf;
 scanf("%d", &sz);
 buf = (char*)alloca(sz);
 gets(buf);
 printf(buf);
}
```

Figure 3. MSVC sample using _alloca function to dynamically create buffer in the stack.

```
__declspec(align(32))
struct S {
 long long a, b, c;
};

void func(void)
{
  struct S s;
  char buf[8];
  gets(buf);
  fill(&s);
  printf("%s %d %d %d\n",
    buf, (int)s.a,
     (int)s.b, (int)s.c);
}
```

Figure 5. Function with 32-bytes aligned local variable.

```
push ebp
mov ebp, esp
sub esp, 8
mov eax, ___security_cookie
xor eax, ebp
mov [ebp-4], eax
push esi
...
call __alloca_probe_16
...
lea esp, [ebp-12]
; Restore esi
pop esi
mov ecx, [ebp-4]
xor ecx, ebp
call @__security_check
mov esp, ebp
pop ebp
ret 0
```

Figure 4. Sample with _alloca compiled by MSVC.

```
push ebx
mov ebx, esp
sub esp, 8
and esp, -32
add esp, 4
push ebp
mov ebp, [ebx+4]
mov [esp+4], ebp
mov ebp, esp
sub esp, 64
mov eax, ___security_cookie
xor eax, esp
mov [ebp-4], eax
...
mov ecx, [ebp-4]
xor ecx, ebp
call @__security_check
mov esp, ebp
pop ebp
mov esp, ebx
pop ebx
ret 0
```

Figure 6. MSVC sample with aligned structures in the stack. ebp is protected by the cookie, but restoring esp from ebx is not.

```
 struct S {
  long long a, b, c;
 }
__attribute__((aligned(32)));
 void func(void)
 {
  char buf[8];
  int n, i;
  gets(buf);
  sscanf(buf, "%d", &n);
  struct S s[n];
  // other stuff
  ..
 }
```

Figure 7. Variable-length array of the aligned structures.

```
push ebp
mov ebp, esp
push ebx
push edi
push esi
and esp, 0xffffffe0
sub esp, 0x60
mov esi, esp
mov eax, ___stack_chk_guard
mov [esi+0x48], eax
...
mov eax, ___stack_chk_guard
cmp eax, [esi+0x48]
jne L
lea esp, [ebp-0xc]
pop esi
pop edi
pop ebx
pop ebp
ret
L: call ___stack_chk_fail
```

Figure 8. Potentially unsafe code generated by clang. esi acts as a frame pointer, esp is loaded from ebp that could be corrupted by unsafe nested functions.

We investigated newer Microsoft C compilers and found that security checks now are enabled when `_alloca` is used, but these compilers place dynamically allocated buffers directly below the callee-saved registers (Figure 2).

It means that callee-saved registers may be overwritten with buffer overflows. Consider the source code in Figure 3. It dynamically allocates memory in stack. This buffer is placed on the top of stack frame, below the callee-saved registers (Figure 2).

When `_alloca` is not used, compiler saves registers from overwriting by placing them at the top of the stack. But stack protection in MSVC lacks complete protection when `_alloca` function is used.

Figure 4 shows the compiled code. `ebp` and `esp` are protected by the security cookie, because `ebp` value is used to check the cookie, and `esp` is restored from `ebp`. `esi` is placed above the security cookie. And buffer allocated by `_alloca` is placed below saved `esi` in the memory. Therefore `esi` is not protected by being overwritten with buffer overflow.

Callee-saved registers are not read until function exit. Therefore this problem may be solved by moving calleesaved registers deeper in stack to protect them with the security cookie.

Another MSVC sample is presented in Figure 5. Function in this sample allocates aligned variable in the stack. The compiler produces the code presented in Figure 6. It aligns `esp` value to 32 bytes boundary which allows storing aligned structures in the stack. The compiler needs an additional register to store original unaligned `esp` value. It uses `ebx` register for that. This is callee-saved register and it is meant to remain unchanged in nested function calls.

But restoring `esp` from `ebx` at function exit is not protected by the security cookie. Therefore one can attack `esp` through corrupting `ebx` in nested functions. These functions may be located in other modules and therefore may be compiled without stack protection, allowing stack overflow exploitation. Or it may have vulnerability described in the beginning of this section.

Using ”omit frame pointer“ compilation option does not fix this example. Compiler uses `ebp` to store original `esp` value, but this `ebp` value is not verified in security check in the epilog. In this case attacker may hijack the stack pointer by overwriting `ebp` in unsafe nested function.

4.3. clang 3.7.1

Stack frame used by clang 3.7.1 compiler differs from MSVC. It puts callee-saved registers under the protection of the canary value. Therefore these values will not be popped back, because security check will stop the execution.

We checked all prepared samples and discovered that stack protection in this compiler could be unsafe in programs with aligned variable-length arrays. Consider the sample in Figure 7. Stack frame produced by clang is presented in Figure 2. It included unaligned callee-saved registers and aligned local variables.

clang generates the assembly code presented by Figure 8. It uses two copies of a stack pointer. One is stored in `ebp` and used to restore `esp` at exit. Another is located in `esi` and used as an aligned frame pointer.

Omitting frame pointer doesn't work here, because variable-length array size is unknown at compile time. Imagine that this function calls some unsafe code which corrupts `esi`. Then checking of the canary value will not be successful. Therefore frame pointer is also protected by stack guard together with stack frame contents.

But stack guard does not protect `ebp`. `ebp` is used to restore `esp` in the epilog. Therefore if there exists suitable unsafe code invoked by this function (e.g., library compiled without stack protection), attacker may overwrite stack and frame pointers.

How to eliminate vulnerability of the stack pointer? Compiler may place saved `esp` value in the aligned stack frame above the canary value. In this case the attacker couldn't overwrite saved `esp` value without being noticed.

4.4. GCC 5.2.0 and 4.6.2

gcc uses all protection methods described above: stack canaries, variable reordering, and copying function arguments.

Scheme of stack protection in gcc differs from the one used in MSVC (Figure 2). Saved registers are protected by the canary value. Therefore the user cannot attack stack pointer and saved registers when stack protection is enabled.

We investigated all examples and didn't found any cases where saved registers, stack pointer, or frame pointer could be hijacked. gcc does not allocate multiple frame pointers. Therefore it is not vulnerable to attack which corrupts one of them. And all saved registers are protected by the canary value.

5. Attack Vectors

In previous section we described flaws in the security checks of the modern compilers. Code samples described above reveal two possible attack vectors: overwriting stack pointer and overwriting callee-saved registers.

Overwriting `esp` may be used to control program execution through pointing esp at the memory where known address is located. Function will try to return to that address and will jump to the desired code. It could be something like `AuthentificationSuccess` to make ”useful“ work, exit for DoS'ing without an alert, or shellcode supplied by the attacker.

Another kind of attack is hijacking the stack or frame pointer to control local variables. Function works with local variables through the frame pointer. If its value is supplied by the attacker, function may read or modify unattended data. This may lead to leakage of sensitive data or to taking alternative branches in the function. Finally, the function will jump to some address with ret instruction. Richarte in his paper [12] presents an example of such an attack.

Modern operating systems have address space layout randomization (ASLR) enabled, which hampers these kind of attacks, because an attacker cannot guest the target address. However, this protection may be bypassed in some cases. E.g., by using the addresses from a module without ASLR [11].

Third kind of attack is overwriting caller's local variables. Stack pointer in this attack is not affected — application will not crash due to jumping to some incorrect address. Attacker has to corrupt callee-saved registers, exploiting the flaws of stack protection mechanism. Caller will not detect that these registers changed, because calling convention declares that callee preserves register values.

6. Related Work

Other attacks and compiler enhancements are described by other researchers. Wilander et al. states, that attacks on return address and old base pointer can be successfully prevented by the runtime stack protection methods [7], [19]. These protection methods are described in section 2. In our research we verified these results and found just few cases when protection can be bypassed.

Paper ”Four different tricks to bypass StackShield and StackGuard protection“ presents description of ways to bypass the stack protection techniques [12]. It focuses on stack frame hijacking, when it is not protected by the canary. State-of-the-art compilers protect frame pointer as well as return address, therefore it cannot be overwritten as easy as in 2002.

Function argument attack described in that paper can also be fought back by the modern compilers. They copy arguments to the stack area above the buffer. Buffer overflow cannot touch arguments when they are located at the lower addresses.

Our research show that `alloca` makes stack protection task harder for the compiler. `alloca` is already known as a source of stack overflow attacks [20]. Therefore every use of this function has to be double-checked.

Canary value protection bypassing methods using exception handlers, virtual tables and couple of other approaches are presented in [11]. It means that stack protection already has some drawbacks in addition to ones that we discovered.

We haven’t found any flaws in gcc stack protection method. However, we didn’t take in account several overflow possibilities, including overwriting of one local buffer with overflow of the another. Paper [17] describes different approaches of enhancing stack protection in gcc. The first improvement is verification of the canary not only when function returns, but also when the function issues a call to another function. This check prevents passing invalid arguments to the nested function. The second improvement is assigning an individual canary for each buffer. With this patch overwriting of one local buffer with another will be detected by the security check. The third improvement makes canary location and failure probabilistic. It makes application harder to attack and reduces amount of information supplied to the attacker in case of the failure. Authors also present patches that implement described protection enhancements. These patches may incur much greater runtime overhead than current protection methods. Therefore they are not used by default yet.

7. Conclusion

In this paper we presented the analysis of the buffer overflow protection methods used in modern compilers. Our tests showed that modern compiler may miss some cases where stack pointer is not protected. Attacker may get control over the application which stack is “protected” with canary value.

We found that methods used in clang and MSVC have several flaws. Both of these compilers does not protect restoring of `esp` from one of the registers, when using aligned data structures in the stack. MSVC also doesn’t protect all saved registers in programs with `_alloca` function.

gcc was the third compiler to examine. We haven’t found any protection issues in samples compiled with gcc.

Tests that were created in this research are released in repository https://github.com/Dovgalyuk/SecurityFlaws. Everyone can generate assembly files for its own compiler and examine the security features of code generation.

Acknowledgments

The work was partially supported by RFBR, research project No. 14-07-00411 a.

References

[1]. Y. Younan, “25 years of vulnerabilities: 1988–2012,” Tech. Rep., 2012. [Online]. Available: https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf

[2]. M. Vallentin, “On the evolution of buffer overflows,” 2007.

[3]. D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, “Static code analysis to detect software security vulnerabilities - does experience matter?” in Availability, Reliability and Security, 2009. ARES ’09. International Conference on, March 2009, pp. 804–810.

[4]. A. Austin and L. Williams, “One technique is not enough: A comparison of vulnerability discovery techniques,” in 2011 International Symposium on Empirical Software Engineering and Measurement, Sept 2011, pp. 97–106.

[5]. N. Rutar, C. B. Almazan, and J. S. Foster, “A comparison of bug finding tools for java,” in Proceedings of the 15th International Symposium on Software Reliability Engineering, ser. ISSRE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 245–256. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2004.1

[6]. H. Sun, X. Zhang, C. Su, and Q. Zeng, “Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability,” in Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ser. ASIA CCS ’15. New York, NY, USA: ACM, 2015, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/2714576.2714605

[7]. J. Wilander and M. Kamkar, “A comparison of publicly available tools for dynamic buffer overflow prevention,” in IN NDSS, 2003.

[8]. B. Bray, “Visual studio .net 2003: Compiler security checks in depth,” February 2002. [Online]. Available: https://msdn.microsoft.com/enus/library/Aa290051

[9]. “Stack smashing protector.” [Online]. Available: http://wiki.osdev.org/Stack_Smashing_Protector

[10]. Bulba and Kil3r, “Bypassig stackguard and stackshield,” Phrack Magazine, vol. 56, May 2000. [Online]. Available: http://phrack.org/issues/56/5.html

[11]. C. Team, “Exploit writing tutorial part 6: Bypassing stack cookies, safeseh, sehop, hw dep and aslr,” 2009. [Online]. Available: https://www.corelan.be/index.php/2009/09/21/exploit-writingtutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

[12]. G. Richarte, “Four different tricks to bypass stackshield and stackguard protection,” World Wide Web, vol. 1, 2002.

[13]. A. Sotirov and M. Dowd, “Bypassing browser memory protections,” in In Proceedings of BlackHat, 2008. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf

[14]. A. One, “Smashing the stack for fun and profit,” Phrack Magazine, vol. 49, November 1996. [Online]. Available: http://phrack.org/issues/49/14.html

[15]. klog, “The frame pointer overwrite,” Phrack Magazine, vol. 55, September 1999. [Online]. Available: http://phrack.org/issues/55/8.html
[16]. 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 In Proceedings of the 7th USENIX Security Symposium, 1998, pp. 63–78.
[17]. A. Seredinschi, Drago¸s-Adrian; Sterca, “Enhancing the stack smashing protection in the gcc,” Studia Universitatis Babe¸s-Bolyai, Informatica, vol. LV, Number 1, 2010.
[18]. Y. WU, “Enhancing security check in visual studio c/c++ compiler,” in Software Engineering, 2009. WCSE ’09. WRI World Congress on, vol. 4, May 2009, pp. 109–113.
[19]. P. Silberman and R. Johnson, “A comparison of buffer overflow prevention implementations and weaknesses.” [Online]. Available: https://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf
[20]. C. Evans, “glibc alloca() memory corruption,” 2011. [Online]. Available: https://packetstormsecurity.com/files/98720/

Когда защита стека в компиляторах не срабатывает?

Павел Довгалюк <pavel.dovgaluk@ispras.ru>
Владимир Макаров <vladimir.makarov@novsu.ru>
Новгородский государственный университет,
173003, Россия, Великий Новгород, ул. Большая Санкт-Петербургская, д. 41

Аннотация. Основная часть уязвимостей в программах вызвана переполнением буфера. Чтобы предотвратить переполнение буфера и уменьшить ущерб от него используется безопасное программирование, аудит исходного кода, аудит бинарного кода, статические и динамические особенности кодогенерации. В современных компиляторах реализованы механизмы защиты, работающие на этапе компиляции и на этапе выполнения скомпилированной программы: переупорядочивание переменных, копирование аргументов и встраивание стековой канарейки. В статье описывается исследование, посвященное поиску недостатков в этих механизмах. Мы протестировали компиляторы MSVC, gcc и clang и обнаружили, что два из них содержат ошибки, позволяющие эксплуатировать переполнение буфера при определенных условиях.

Ключевые слова: переполнение буфера; стековая канарейка; gcc; msvc; clang

DOI: 10.15514/ISPRAS-2016-28(5)-3

Для цитирования: П.М. Довгалюк, В.А. Макаров. Когда защита стека не срабатывает? Труды ИСП РАН, том 28, вып. 5, 2016, стр. 55-72 (на английском). DOI: 10.15514/ISPRAS-2016-28(5)-3

Литература

[1]. Y. Younan, “25 years of vulnerabilities: 1988–2012,” Tech. Rep., 2012. [Online]. Available: https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf
[2]. M. Vallentin, “On the evolution of buffer overflows,” 2007.
[3]. D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, “Static code analysis to detect software security vulnerabilities - does experience matter?” in Availability, Reliability and Security, 2009. ARES ’09. International Conference on, March 2009, pp. 804–810.
[4]. A. Austin and L. Williams, “One technique is not enough: A comparison of vulnerability discovery techniques,” in 2011 International Symposium on Empirical Software Engineering and Measurement, Sept 2011, pp. 97–106.
[5]. N. Rutar, C. B. Almazan, and J. S. Foster, “A comparison of bug finding tools for java,” in Proceedings of the 15th International Symposium on Software Reliability

Engineering, ser. ISSRE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 245–256. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2004.1
[6]. H. Sun, X. Zhang, C. Su, and Q. Zeng, “Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability,” in Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ser. ASIA CCS ’15. New York, NY, USA: ACM, 2015, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/2714576.2714605
[7]. J. Wilander and M. Kamkar, “A comparison of publicly available tools for dynamic buffer overflow prevention,” in IN NDSS, 2003.
[8]. B. Bray, “Visual studio .net 2003: Compiler security checks in depth,” February 2002. [Online]. Available: https://msdn.microsoft.com/enus/library/Aa290051
[9]. “Stack smashing protector.” [Online]. Available: http://wiki.osdev.org/Stack_Smashing_Protector
[10]. Bulba and Kil3r, “Bypassig stackguard and stackshield,” Phrack Magazine, vol. 56, May 2000. [Online]. Available: http://phrack.org/issues/56/5.html
[11]. C. Team, “Exploit writing tutorial part 6: Bypassing stack cookies, safeseh, sehop, hw dep and aslr,” 2009. [Online]. Available: https://www.corelan.be/index.php/2009/09/21/exploit-writingtutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/
[12]. G. Richarte, “Four different tricks to bypass stackshield and stackguard protection,” World Wide Web, vol. 1, 2002.
[13]. A. Sotirov and M. Dowd, “Bypassing browser memory protections,” in In Proceedings of BlackHat, 2008. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf
[14]. A. One, “Smashing the stack for fun and profit,” Phrack Magazine, vol. 49, November 1996. [Online]. Available: http://phrack.org/issues/49/14.html
[15]. klog, “The frame pointer overwrite,” Phrack Magazine, vol. 55, September 1999. [Online]. Available: http://phrack.org/issues/55/8.html
[16]. 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 In Proceedings of the 7th USENIX Security Symposium, 1998, pp. 63–78.
[17]. A. Seredinschi, Drago¸s-Adrian; Sterca, “Enhancing the stack smashing protection in the gcc,” Studia Universitatis Babe¸s-Bolyai, Informatica, vol. LV, Number 1, 2010.
[18]. Y. WU, “Enhancing security check in visual studio c/c++ compiler,” in Software Engineering, 2009. WCSE ’09. WRI World Congress on, vol. 4, May 2009, pp. 109–113.
[19]. P. Silberman and R. Johnson, “A comparison of buffer overflow prevention implementations and weaknesses.” [Online]. Available: https://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf
[20]. C. Evans, “glibc alloca() memory corruption,” 2011. [Online]. Available: https://packetstormsecurity.com/files/98720/