Advanced Threat Research Innovation Data Fortify

Technical White Paper

October 2022

Revision: 0.5

Hasarfaty, Shai - Principal Offensive Security Research Engineer

Document Number: 749074

Contents

Revision History

Document Number	Revision Number	Description	Revision Date
749074	0.5	• Initial release	October 2022

§§

1 Scope and Product Target

This document focuses on the prevention of buffer overflow or data leakage that might happen due to misuse/missing size input validation prior to passing it to the dangerous standard C library function such as: memcpy, strcpy, memmove, sprint, strlen, strcat, strncat etc. The size argument is lacking input validation that corresponds to the actual size of the given address. This lack of input validation can lead to a vulnerability that will cause linear data corruption or information leakage.

The main assumption of this solution is that pointers (destination or source) are not corrupted. This assumption was made after many reviews of past discovered issues and their root-cause. Refer to Motivation for more details.

The work done in this paper is targeting a solution for x86 Intel architecture (32bit) and on ELF file format and on GCC compiler tool chain. The solution is specifically designed for embedded systems that have limited resources such as flash size, constrained runtime execution and memory consumption but can fit to other OSs and environments.

This solution is planned to be integrated into Intel® Converged Security and Management Engine (CSE / CSME) as a defense in-depth mitigation in addition to its existing (refer to number [24] in References) exploitation mitigation.

§§

2 Motivation

There is a clear indication that one (of many) methods of work to discover vulnerabilities by bug-hunters, is to focus on finding data corruption or leakage that can occur on the heap, stack or global variables. These types of vulnerabilities could lead to arbitrary code execution, control of execution flow, or disclosure of sensitive runtime data in order to find these "low-hanging-fruit" issues, one of the most popular method is to perform static analysis via reverse engineering. First, bug-hunters need to find all the standard C library functions, such as: malloc, free, memcpy, read, strcpy etc. Once this stage is done, the bug-hunters can start to automate the process of finding areas with high probability (refer to number [11] in References) to find issues related to missing input validation on which to focus their efforts.

This automated process has been presented multiple times in conferences (refer to numbers 15 and 16 in References). In some cases, tools were shared to ease the automation for this type of work using frameworks such as: NSA Ghidra (refer to number [1] in References), IDA and Binary-Ninja (refer to number [2] in References).

Observing the latest vulnerabilities that were discovered in many products strengthen the assumption that vulnerabilities related to standard C library function use are prevalent:

- Exploiting Wi-Fi Stack on Tesla Model S (refer to number [3] in References)
    - Memcpy without size validation - copying the data from the ADDTS response packet to the HostCmd_CMD_WMM_ADDTS_REQ structure. The length of the copied data is calculated by subtracting 4 bytes length of action header from length of action frame. However, if the action frame only contains a header, and the length of the header is only 3 bytes, the length that needs to be copied is 0xFFFFFFFF. This can eventually override a function pointer used in interrupt handler to call the right function to handle such large overflow. As a consequence, it causes code execution via function pointer control.
- Instagram RCE: Code Execution Vulnerability in Instagram App (refer to number [4] in References)
    - This vulnerability refers to usage of wrong malloc function that gets corrupted on a second stage via the memcpy function - The allocated size is calculated by multiplying the image's width, height and output components. Those sizes are unchecked and in our control. When abused, they can lead to an integer overflow. Conveniently enough, this buffer is then passed to memcpy function, leading to a heap-based buffer overflow.
- 17 Years-old Bug in Windows DNS Servers (refer to number [5] in References)
    - This vulnerability refers to an integer truncation that causes a wrong memory allocation and corruption via usage of memcpy function– The Integer overflow leads to a much smaller allocation than expected. The allocated memory address is then passed as a destination buffer for memcpy, leading to a Heap-Based buffer overflow.
- Intel Management Engine (ME) (refer to number [6] in References)
    - File read into buffer on the stack without verification of the file size that will match the destination size causing a stack buffer overflow.

    - Overflow was large and changed data in other frames of the stack that contained pointers that was used after the overflow that allowed the corruption to be transformed to a write-what-where and from that to code execution.
- "BootHole" vulnerability in the GRUB2 bootloader (refer to number [7] in References)
    - This vulnerability refers to usage of strcpy function used without destination size verification – this is due to not implementing ERROR handle macro to stop execution due to FATAL_ERROR. Since no error handle was made the code just print to log that token is too large and continue to use "strcpy".
- Ripple20 - 19 Zero-Day Vulnerabilities Amplified by the Supply Chain (refer to number [8] in References)
    - This vulnerability happens in the Treck network stack. Demonstrated info leak from the heap due to wrong size usage of the source pointer during memcpy. Demonstrated heap buffer overflow due to wrong calculation of destination size while using memcpy
- Don't be silly – it's only a lightbulb (refer to number [9] in References)
    - Multiple heap overflows due to bug-hunter controlled size that is not validated correctly during "memcpy"

Another popular automation method is by fuzzing (refer to number [20] in References) the product entry points and observe if the product has crashed.

In some of the demonstrated vulnerabilities, it was shown that mitigations such as canary or even shadow stack, may not be useful since they are being enforced/verified only during function epilogue. Since the corruption has already occurred, the corrupted data is being used in the code flow. This allows the exploit developer to shape the data used in the code flow and trigger artificially more issues without triggering the mitigation or detection and bypass them. For example, in shadow stack, we can simply put the same return address but data will be corrupted and code flow will be modified and create data orientation exploitation (refer to number [21] in References).

Since the most used functions in application are the linear copying functions and format string (refer to number [11] in References), it's only a matter of statistics that there will be a bug in one of the usages of these dangerous standard C library functions.

This is the motivation to find a solution that will be able to prevent linear buffer overflow or info-leak from happening before it's too late.

§§

3 Existing Solutions

Today, the most popular solution for catching linear overflows is a compiler option called Address Sanitizer (refer to number [12] in References). The main issues with this solution are performance, code size increase and large memory usage. Address sanitizer creates a shadow memory of the entire process needs to instrument every buffer creation and to represent the size allocation in memory. This requires large memory space just for the runtime data and code. This solution has a slower (x4) runtime performance caused by the need to instrument every instruction that leads to data read or data write in the code flow (not only the standard C library functions). Another issue with this solution is the lack of support for other CPU architectures (currently supported only on x86_64 and ARM and ARC) and it can work only if using glibc (GNU C Library). Therefore, address sanitizer is not applicable in production code and mainly used during product testing.

Another solution that exists is Valgrind (refer to number [13] in References). The main issue with Valgrind is that it was designed as a tool for memory debugging, memory leak detection, and profiling and not for production code and its performance overhead is x40 over the original code.

Yet another compiler option is, FORTIFY_SOURCE (refer to number [14] in References). This option tries to give a lightweight runtime protection to some memory and string standard C library functions. FORTIFY_SOURCE replaces the standard C library string and memory functions with their "*_chk" counterparts (i.e.: memcpy_chk). In the new implementation the compiler is passing an additional argument that is the original size of the statically allocated variable. The limitations of this implementation are scope and pointer aliasing. It can only work if the function that is calling the actual standard C memory and string library functions is within the same scope of the function declared local variables (same translation unit). If the locals are passed to a nested function call that calls the standard C library function, it can work only if the compiler can optimize the code of that function as inline, making the local variable part of the scope of the standard C library functions.

FORTIFY_SOURCE does not cover dynamic allocation (heap) out-of-bounds access and it cannot handle pointer aliasing to existing global or local variables (like passing them as an argument), only if used directly. FORTIFY_SOURCE like address sanitizer, requires glibc in order to work and cannot be customized to be applied on more functions beside the pre-defined standard C library memory and string functions.

In the past Intel® MPX technology (Memory Protection Extension) gave some level of a solution to stack locals but due to high impact on code size and performance (refer to number [22] in References), it was removed in future roadmap of Intel CPUs and is no longer supported from GCC 9 and onward.

An optional way to prevent the misuse of sizes in the standard C library functions is by banning (refer to number [17] in References) the use of these dangerous/un-safe functions and use safe function that are more "developer aware" library functions. The safe functions do not only require the size of the buffer to copy, but also require the size of the destination and the source (i.e.: memcpy_s) or limit the size of the data being copied (i.e.: strncpy_s). Inside of these functions, the required destination and source sizes are compared to not copy beyond the stated sizes. However, these implementations are still depended on the developer to pass the right sizes. If the wrong sizes are given, then buffer overflow still occurs (refer to number [18] in References).

§§

4 Proposed Solution

The proposed solution is to implement a function that will be able to retrieve, for a desired address, it's real allocation size in runtime and to apply it in every function we choose to. This way, we can get to a full customization of our code and not only in a pre-defined location. This can work either as a wrapper for existing functions, or inside of them to be transparent to the developers. Example:

```
void memcpy(void *dst,void *src ,size_t size)
{
    #ifdef VERIFY_DST
        verify(dst,size);
    #endif
    #ifdef VERIFY_SRC
        verify(src,size);
    #endif
    //body impl of memcpy
}
```

In case the sizes do not match, then the verification function can either not return and halt/invoke process termination (in some firmware cases - firmware reset). Or, like "memcpy_s" will return status error for error handling and continue execution flow correctly and will not cause a denial of service. Example:

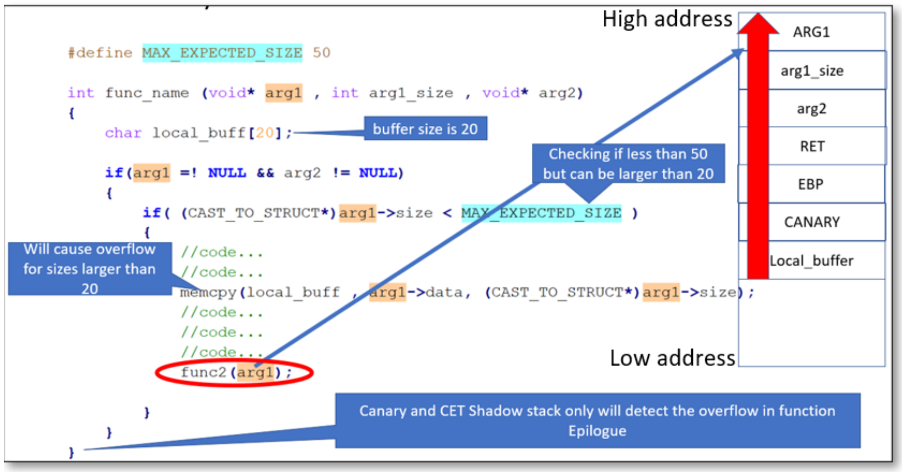
```
ERROR_STATUS memcpy_warpper(void *dst,void *src ,size_t size)
{
    #ifdef VERIFY_DST
      if( !verify(dst,size ) )
            return ERROR;
    #endif
    #ifdef VERIFY_SRC
      if( !verify(src,size) )
            return ERROR;
    #endif
    //body impl of memcpy
    return SUCCESS;
}
```

§§

5 Implementation Details

5.1 Stack Size verification – Stack Fortify

Refer to the following image to view an example of a stack overflow that will take advantage of a bug in the buffer size validation. It will corrupt data in "arg1" on the stack and pass it to a new function while it's corrupted and not being detected with the canary or CET (refer to number [25] in References) shadow stack. Potentially in "func2" there could be a bigger issue since "func2" trusts the input verification that was done before and will use the data without any verification. Because the data in "arg1" is completely controlled by the user inputs the likelihood of exploitation success is high.



```
#define MAX_EXPECTED_SIZE 50

int func_name (void* arg1 , int arg1_size , void* arg2)
{
    char local_buff[20];

    if(arg1 =! NULL && arg2 != NULL)
    {
        if( (CAST_TO_STRUCT*)arg1->size < MAX_EXPECTED_SIZE )
        {
            //code...
            //code...
            memcpy(local_buff , arg1->data, (CAST_TO_STRUCT*)arg1->size);
            //code...
            //code...
            //code...
            func2(arg1);

        }
    }
}
```

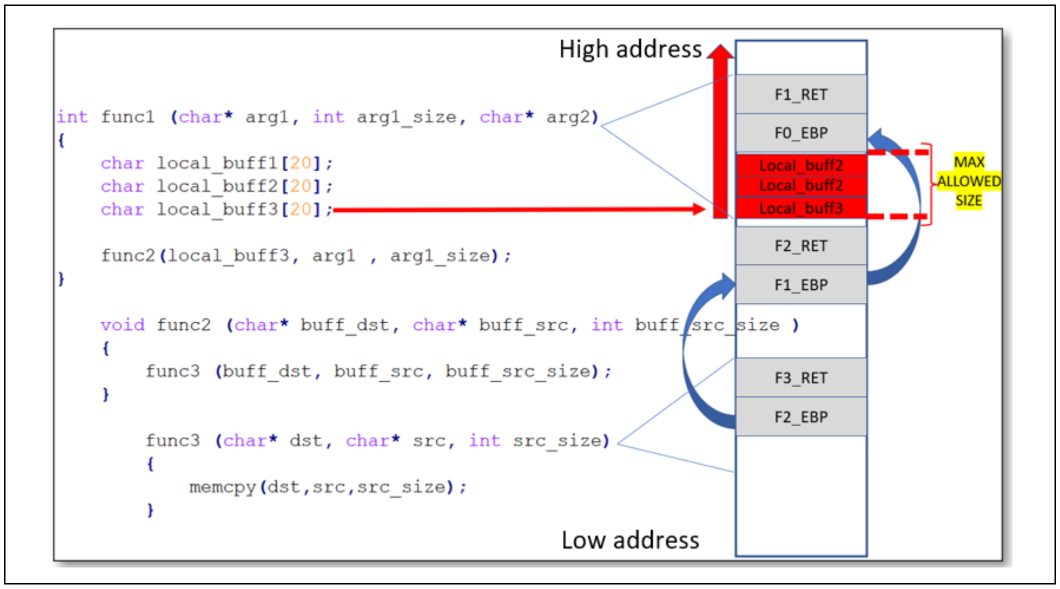
High address
ARG1
arg1_size
arg2
RET
EBP
CANARY
Local_buffer
Low address
buffer size is 20
Checking if less than 50 but can be larger than 20
Will cause overflow for sizes larger than 20
Canary and CET Shadow stack only will detect the overflow in function Epilogue

To solve this issue and get the size of "local_buff" we leverage a feature of compilers that embed data that allows debuggers the ability to get function back tracing information. We use this same data to gain runtime knowledge of the function stack frame size.

Because the debugger needs to know the call-trace, the assembly code is generated in a way that allows generation of the call trace. Since EBP points to the caller stack frame (old EBP), EBP+4 holds the return address.

By recursively reading EBP and its pointer to the previous stack frame location, we can find the address of the frame pointer (EBP) where the desired destination address is lower than the previous stack frame location. With this knowledge the calculation of the maximum size allowed for copying can be accomplish.

The following picture illustrates the trace that needs to be implemented to verify the maximum allowed copy/read size.



```
int func1 (char* arg1, int arg1_size, char* arg2)
{
    char local_buff1[20];
    char local_buff2[20];
    char local_buff3[20];

    func2(local_buff3, arg1 , arg1_size);
}

    void func2 (char* buff_dst, char* buff_src, int buff_src_size )
    {
        func3 (buff_dst, buff_src, buff_src_size);
    }

        func3 (char* dst, char* src, int src_size)
        {
            memcpy(dst,src,src_size);
        }
```

To make sure this structure on the stack will be generated a compiler option "-fno-omit-frame-pointer" MUST be used.

In the example above, since we can "backtrace" to a higher address than "local_buf_3", we know the frame limits. Calculation of the max-allowed copy size in the origin stack frame of the variable, prevents from copying beyond the function frame:

TRACED_HIGHER_EBP – PTR_LOCAL_BUF_3 = MAX_ALLOWED_SIZE

This still allows an attacker to overrun frame inner local variables (Refer Appendix for solution side notes) but the likelihood of successful exploitation will be lower significantly. Stack canary/protection is very important since it is not only used as a detection of corruption that can occur due to other vulnerability types but If stack protection is enabled, the compiler (GCC) defers all stack variables, so that the compiler can re-order the strings to the top of the frame (GCC source file cfgexpand.c, function "defer_stack_allocation"). This means, the buffers will be located higher on the stack than the primitive local variables (integers etc.)

5.2 Global Size Variables Verification – Globals Fortify

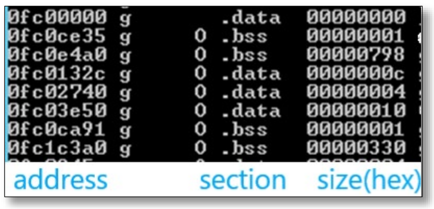
For global variables protection we will focus only on BSS (un-initialized variables) and DATA (initialized variables) sections. RODATA section is not relevant because:

1. RODATA is a known static fixed value and does not change – Therefore, info leak is not relevant.
    a. A mitigation for the case of a large info-leak that can go beyond RODATA, is to put a page guard without an access attribute (R/W/X) after RODATA in order to trigger a page fault by the CPU.
2. RODATA should be mapped by the OS as read-only pages in memory. Writing to this memory triggers a page fault and exception.

In order to retrieve the size of a desired global variable in BSS or DATA section, we need to generate metadata post linking that will be used during runtime.

The metadata holds the address and size of each global variable. The way to retrieve the address and size of each global variable is via a binutils application called "objdump".

The global variables symbol information is extracted from the binary by calling "objdump -t" to get the following results.



```
0fc00000 g         .data  00000000
0fc0ce35 g       O .bss   00000001
0fc0e4a0 g       O .bss   00000798
0fc0132c g       O .data  0000000c
0fc02740 g       O .data  00000004
0fc03e50 g       O .data  00000010
0fc0ca91 g       O .bss   00000001
0fc1c3a0 g       O .bss   00000330
```

Using the linker, a new global variable is created in the RODATA section with the required size to hold the entire global variables metadata information:

GLOBAL_VARS_METADATA_SIZE = 4 + ((BSS_AMOUNT + DATA_AMOUNT) * 8)

The first 4-bytes hold the size of GLOBAL_VARS_METADATA_SIZE and the rest is the list. Each entry holds a 32-bit address and a 32 bit size, for a total of 8-bytes per-entry.

Filling the GLOBAL_VARS_METADATA with the required data can be done with either method below:

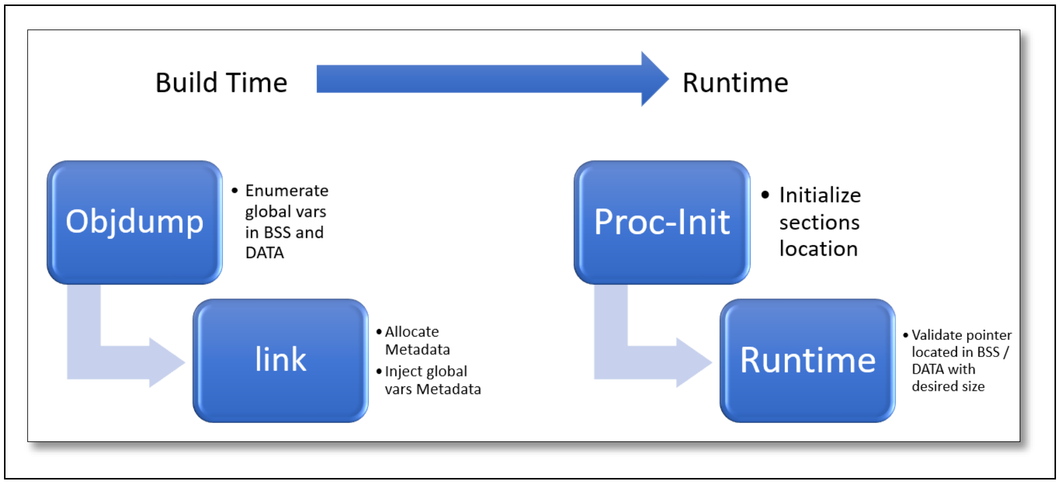
1. usage of "FILL" command of a linker script
2. manually opening the binary file for write and writing to the GLOBAL_VARS_METADATA offset in the file with the values of the address and size of each global variable

Global variables with size of 0, should be ignored since they are only offset reference symbols. For the runtime search to be fast, the entries must be sorted by address in ascending order.

If a given pointer is in the range of BSS/DATA, all that is needed in order retrieve the size of the desired global variable is to apply a binary-search algorithm (flat not recursive) on GLOBAL_VARS_METADATA.

- The search needs to find not only beginning of a pointer but also mid-pointers original size.
- Once a pointer size has been retrieved, verification of size is possible.

The overall illustration of implementation is as follows:


Build Time
Runtime
Objdump
• Enumerate global vars in BSS and DATA
link
• Allocate Metadata
• Inject global vars Metadata
Proc-Init
• Initialize sections location
Runtime
• Validate pointer located in BSS / DATA with desired size

5.3 Heap Size Verification - Heap Fortify

Without diving too deeply into the heap manager implementation, we will focus only on the metadata implementation and will not discuss other heap manager implementation aspects.

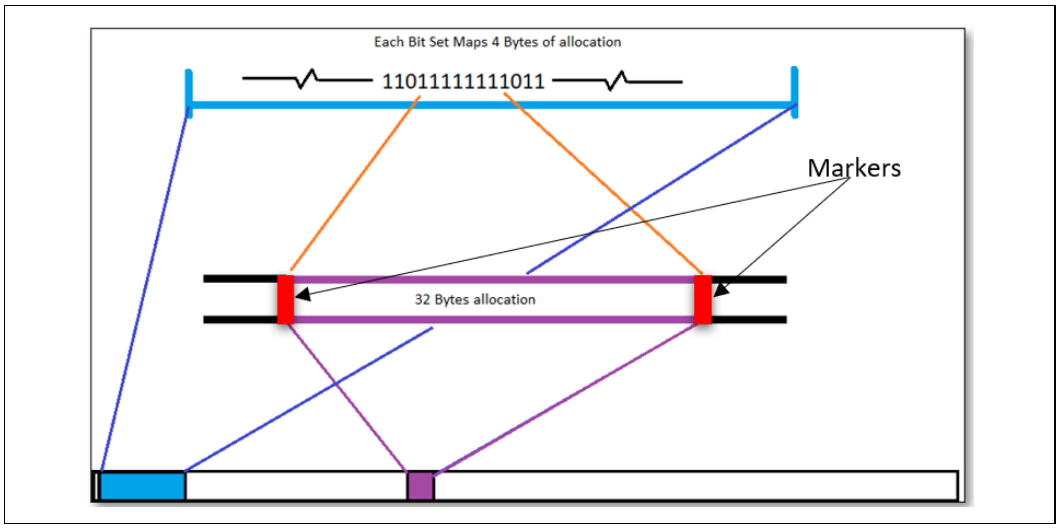
The following section will present details of a specific heap implementation. This does not mean it is the only way to implement a heap manager with metadata, nor that it is the most optimized, but rather an example. This heap implementation is based on a heap that is aligned to a 4-bytes allocation size.

The CSME has many security requirements. The requirement standing out as the most relevant for this section is: Heap Manager shall provide a mechanism of free "bad pointer" detection

The requirement mandates that the Heap Manager prevents freeing a bad pointer by holding metadata that indicates the pointer allocation start and end. If the pointer is pointing to a "free" area, the heap will avoid the free action (freeing un-allocated area is not allowed).

For every page in the pool there is a page descriptor. The page descriptor (PAGE_INFO) is a 16-bit value. The PAGE_INFO contains bit field information about current state of the page managed by the memory manager. For every pool there is an array of such values. Pages are divided into "Free", "Fragmented" (small allocation – less than 4K) and "Segmented" (big allocation - equal to or greater than 4K).

In case a page is marked as a "Fragmented" allocation page, in the beginning of the page itself, there is a bitmap that describes all allocations in the page. The bitmap size is 128-byte (1024-bit) where each set bit ("1") describes a 4-byte allocation. Between each allocation there is a marker of 4-bytes that is marked as "free" (0b) in the byte allocation bitmap. This 4byte marker holds metadata for debugging and overflow detection (marker is verified only during "free" operation). The marker is split into two 16bit parts. The first 16 bits hold the caller information, and the second 16-bits hold a random cookie. The following picture represents this concept:



If a pointer to the heap points to mid-allocation (offset), we can calculate the location of the representation of the address in the bitmap and search for the beginning or end of the pointer and calculate the size of the allocation to be compared against the input we need to validate.

Since bit scanning is a highly optimized CPU instructions ("BSF/BSR"), the performance for such search has low performance impact.

§§

6 Solution Performance Impacts

The performance impact of the different solutions varies due to different approaches for retrieving the size available for a given address.

6.1 Heap Size Retrieval Performance Impact

In case of the heap implementation the code size increase to support heap variable size retrieval is negligible since the instrumentation is only applied in one area and does not need to be applied in every part of the code. The only performance that is relevant is the runtime impact.

In the case of heap addresses it was shown that in case of a small or large allocation the operations that are required are:

1. Get the page index: O(1)
2. Get the classification of the page: O(1)

Small allocation - Bitmap scanning:

The worst case for finding the remaining size allowed for copy is - O(n) where n = SIZE_OF_ALLOCATION / 4.

As stated, even though the search is an O(n), the search is optimized by using the bit scanning instruction (BSF/BSR). The timing of the instruction (refer to number [19] in References) is different between CPU generation and 32bit/64bit architecture.

Big allocation mapping requires searching in each page pool in the allocation page descriptor:

The worst-case for finding the remaining size allowed for copy is - O(n) where n = SIZE_OF_ALLOCATION / 4k.

Meaning, we need to iterate in a granularity of 4K until we find the page that is the last page of the allocation to be able to calculate the allowed access size.

6.2 Global Variables Size Retrieval Performance Impact

In the case of global variables, there is size impact on the read-only data section (RODATA) since it is required to hold the table of all global variable address and size. meaning the size increase is of the executable image and not the code size.

The increase of the RODATA section is calculated as follows:

GLOBAL_VARS_METADATA_SIZE = 4 + ((BSS_AMOUNT + DATA_AMOUNT) * 8)

The runtime impact is a binary search of log (base 2). Hence, worst-case scenario of searching for required address is O(log(n)) where n = BSS_AMOUNT_OF_GLOBAL_VARS + DATA_AMOUNT_OF_GLOBAL_VARS. For example: Given 1000 entries (each entry holds address and size) of global variables will takes about 10 iteration in worst-case scenario to find the size of the variable given an arbitrary address.

By using the compiler options correctly it can be found that binary search are CPU cache friendly and faster results can be achieved if used properly (refer to number [23] in References).

6.3 Stack Variables Size Retrieval Performance Impact

The performance impact is mainly on runtime and is defined by the amount of nested calls. Therefore, the runtime performance impact in the worst-case scenario is O(n) where n = NESTED_CALLS_TREE_DEPTH.

In our experiments we found that in practice, real applications usually have a maximum nested tree depth of n=10, while the average nested tree depth is n=3.

There is also potential code size impact due to the used "-fno-omit-frame-pointer" compilation flag. This is harder to measure and depends on the existing code structure of each product and can varies between one to another.

§§

7 Summary

The solution is not entirely coupled with hardware architecture and the modification required for stack frame size retrieval to support other CPU architectures is very lightweight since it's an already existing implementation in every architecture compiler through the backtrace command, which is implemented in the gdb debugger, since the compilers create the code structure to make production code debug-able and backtrace ready.

Existing industrial solutions have large impact on:

- runtime due to the increase of code size after applying instrumentation of every pointer access in the code flow (i.e.: Address Sanitizer)
- memory usage due to the need to hold large metadata as shadow.

Some solutions were lacking the ability to work with real life code that requires to get the size of an address even from aliased pointers or arguments passed to deeply nested calls (i.e.: FORTIFY_SOURCE).

Preventing the buffer overflow from happening allows programs to continue execution as no corruption occurs. Implementing proper error handling further prevents DoS (denial of service) caused by protections such as stack canary and stack shadowing. If such DoS is not a factor in the product security, the implementation can choose to raise an exception and terminate the process instead.

"Safe API" were introduced by adding the "_s" to the standard C library functions (e.g.: memcpy_s etc.). But safe does not mean secure, and these APIs can be still vulnerable to developers passing wrong sizes (by mistake).

The main limitation of the "Data Fortify" solution is: Internal overflow of data structures within the allocation size. Data Fortify solution is not identifying such overflows. This is the case for heap, global variables, and stack (function frame allocated size). Even with this limitation, the proposed fortification significantly reduces the likelihood of a successful exploitation.

For the Intel® CSE/CSME, data fortify will be applied on the following standard C library functions":

- Memcpy
- Memmove
- Memset
- Strcpy
- Strncpy
- Strcat
- Strncat
- Read
- Strlen
- Strnlen
- Sprintf
- Snprintf

In addition, the verification function was added to some private Intel® CSE APIs that might present a risk as well.

In summary, this paper introduces a solution that is flexible and well balanced between size and runtime impact. Also, Data Fortify can be enabled in production and not only in debug/testing environments The solution is applicable not only on pre-defined functions and can be placed in any code that poses a risk.

The main different concept of this solution to other existing anti-exploitation mitigations is by preventing \ limiting the memory corruption from happening. Unlike most other solutions that are more focused on post-corruptions mitigations that tries to lower the impact or the likelihood of a successful exploitation. Since many types of other vulnerabilities classes exists, we cannot drop any other existing systemic mitigations.

§§

8 References

[1] - Alexei bulazel - Working with Ghidra's p-code to identify vulnerable function calls, 2019

https://www.riverloopsecurity.com/blog/2019/05/pcode/

[2] - https://github.com/trailofbits/binjascripts/blob/master/abstractanalysis/binja_memcpy.py

[3] – Keen Security Lab - Exploiting Wi-Fi Stack on Tesla Model S , 2020

https://keenlab.tencent.com/en/2020/01/02/exploiting-wifi-stack-on-tesla-model-s/

[4] – Gal Elbaz - "#Instagram_RCE: Code Execution Vulnerability in Instagram App for Android and iOS", 2020

https://research.checkpoint.com/2020/instagram_rce-code-execution-vulnerability-in-instagram-app-for-android-and-ios/

[5] – Sagi Tzadik, SIGRed – Resolving Your Way into Domain Admin: Exploiting a 17 Year-old Bug in Windows DNS Servers , 2020

https://research.checkpoint.com/2020/resolving-your-way-into-domain-admin-exploiting-a-17-year-old-bug-in-windows-dns-servers/

[6] – Mark Ermolov , Maxim Goryachy - Positive Technologies, How to Hack a Turned-Off Computer, or Running Unsigned Code in Intel Management Engine , 2017

https://www.blackhat.com/docs/eu-17/materials/eu-17-Goryachy-How-To-Hack-A-Turned-Off-Computer-Or-Running-Unsigned-Code-In-Intel-Management-Engine.pdf

[7] – Mickey Shkatov, Jesse Michael - "BootHole" vulnerability in the GRUB2 bootloader, 2020

https://eclypsium.com/wp-content/uploads/2020/08/Theres-a-Hole-in-the-Boot.pdf

[8] – Moshe Kol, Ariel Schön, Shlomi Oberman - Ripple20 - 19 Zero-Day Vulnerabilities Amplified by the Supply Chain, 2020

https://www.jsof-tech.com/ripple20/

[9] – Eyal Itkin - Don't be silly – it's only a lightbulb, 2020

https://research.checkpoint.com/2020/dont-be-silly-its-only-a-lightbulb/

[10] – 2020 CWE Top 25 Most Dangerous Software Weaknesses

http://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

[11]- El Habib Boudjema, Christèle Faure, Mathieu Sassolas, Lynda Mokdad - Detection of security vulnerabilities in C language applications, 2017

https://onlinelibrary.wiley.com/doi/full/10.1002/spy2.8

[12] – Konstantin Serebryany, Derek Bruening, Alexander Potapenko, Dmitry Vyukov - Address Sanitizer, 2012

https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf

[13] – Valgrind

https://www.valgrind.org/docs/manual/valgrind_manual.pdf

[14] – Jakub Jelinek - SOURCE FORTIFY , 2004

https://gcc.gnu.org/legacy-ml/gcc-patches/2004-09/msg02055.html

[15] – Sophia D'Antoine - Be a Binary Rockstar: An Introduction to Program Analysis with Binary Ninja, 2016

https://www.slideshare.net/codeblue_jp/cb16-dantoine-en

[16] - Steven Vittitoe - Reverse Engineering Windows AFD.sys, 2015

https://recon.cx/2015/slides/recon2015-20-steven-vittitoe-Reverse-Engineering-Windows-AFD-sys.pdf

[17] - Michael Howard - "Banned APIs and Sin Within!", 2013

http://www.hackformers.org/wp-content/uploads/2013/07/Banned-APIs-and-Sin-Within.pptx

[18] – Kryptos Logic - RDP to RCE: When Fragmentation Goes Wrong , 2020

https://www.kryptoslogic.com/blog/2020/01/rdp-to-rce-when-fragmentation-goes-wrong/

[19] - Torbj¨orn Granlund, Instruction latencies and throughput for AMD and Intel x86 processors, 2019

https://gmplib.org/~tege/x86-timing.pdf

[20] - Barton Miller, Operating System Utility Program Reliability − The Fuzz Generator, 1988

http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf

[21] - Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, Zhenkai Liang

Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks, 2016

https://www.comp.nus.edu.sg/~tsunami/papers/paper.pdf

[22]- Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, Christof Fetzer - Intel® MPX Explained: A Cross-layer Analysis of the Intel® MPX System Stack, 2018

https://intel-mpx.github.io/code/submission.pdf

[23]- Joaquín M López Muñoz, Cache-friendly binary search, 2015

http://bannalia.blogspot.com/2015/06/cache-friendly-binary-search.html#:~:text=High%2Dspeed%20memory%20caches%20present,behind%20classes%20such%20as%20Boost.

[24]- Intel® Converged Security and Management Engine (Intel® CSME) Security White Paper

https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/intel-csme-security-white-paper.pdf

[25]- Control-flow Enforcement Technology Specification

https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

§§

9 Appendix – Solution Side Notes

1. The invocation to the re-implementation of the library APIs (memcpy etc.) can be done via compiler command line "--wrap" option that will call the "__wrap_symbol" library function and not the "__real_symbol" and the wrapper will call the "__real_symbol".
2. It is needed to not only use the existing system calls wrappers but also find in the product code other places that the implements helper functions that eventually behave like linear write/read and apply the suggested solution on them as well (calling the verification function prior to doing the behavior)
3. To have the ability to check sizes of arbitrary pointers in memory requires the product to be able to pre-know each area and to whom it belongs to. Usually area of memory is divided into sections and we have (not by order):
    a. Code (text)
    b. Stack
    c. BSS
    d. DATA
    e. RODATA
    f. Heap

To be able to apply any protection, the application requires to know in runtime each pointer to what section it belongs to.

4. The proposed solution is not covering inner buffers overflow within the single allocation. Meaning, if a global variable is constructed from multiple types (i.e.: structure), a buffer overflow is considered only when the overflow/leak is beyond the entire structure

An overflow will be considered only if trying to copy beyond original allocation of size. Inner data writes/read are not considered as out of bound access. Stack overflow is considered as overflow for this proposal only if the access out of bounds is beyond the function "Stack Frame Size". The function "Stack Frame Size" is calculated during compilation time. The compiler knows for the required stack frame size for each function and move the stack pointer accordantly, leaving space for the entire the function local variables to be stored on the stack (so the locals are actually between the stack pointers and the return address). An out of bound write/read will be considered an overflow on stack only if the size of copying/reading on the stack is going to be larger than the function frame size allocation

5. If needed during initialization of OS area of memory i.e.: Zeroing all BSS memory area with "memset", it will be required to implement an "not_secure_memset" that will not have the verification option and "bypass" the protection.
6. Unlike the heap fortify and globals fortify, if the variable is located on the stack (function locals) it is not possible to "getAllocedSize" of an arbitrary pointer of the stack without it being passed as nested argument on the stack. In case of heap and global variables, since they are described in metadata, it is possible request size of the desired pointer that points to arbitrary address of heap/global memory sections.

In case heap implementation the right way to implemented it is by return value of "success" or "failure" in case that the pointer points to a free area, and the return of size will be return as reference function parameter/argument.

7. Instead of verifying both destination and source pointers, performance impact can be improved by only choosing to verify for destination copying to eliminate buffer overflow but taking the risk on having info-leak. Also, another way to customize the performance impact is by profiling your product and what is more at risk by observing the product past issues and focus the verification function only on these areas. For example, let's say that we have less risk in info leak of global variables, but all the rest is still a risk. We can define the prototype to pass a BIT_SET to the only desired verification

```
ERROR_STATUS memcpy_warpper(void *dst,void *src ,size_t size)
{
    #ifdef VERIFY_DST
      if( !verify(dst,size, BIT_SET_STACK | BIT_SET_HEAP| BIT_SET_GLOBALS ) )
            return ERROR;
    #endif
    #ifdef VERIFY_SRC
      if( !verify(src,size ,  BIT_SET_STACK | BIT_SET_HEAP) )
            return ERROR;
    #endif
    //body impl of memcpy
    return SUCCESS;
}
```

§§