Nostalgic memory - An attempt to understand evolution of memory corruption mitigations

Author: Shubham Dubey

Arbitrary Code execution due to memory corruption is not new. It's been there since forever(to be more precise, probably since the 1980s) and still a major thing. For instance,70 percent of all security issues addressed in Microsoft products are caused by violations of memory safety. Similar proportion holds true for linux. However, a lot has changed over the years. The conventional attacks and exploits that were prevalent a decade ago no longer hold the same significance. Simultaneously, new protective measures have been introduced periodically to mitigate these attacks, but have also inadvertently led to the discovery of novel attack techniques to circumvent these mitigations. For a diligent security researcher or application developer, It's hard to follow this ongoing cat and mouse race. While certain protective measures are widely recognized, there are many others that are not well-known yet possess intriguing narratives or implementation details. This realization has inspired me to document all these protections that I have encountered or become aware of throughout my years of study, and to ponder why I had not been acquainted with them earlier.

How it all started (The chaos)

> “*At the beginning there was only Chaos, Night, dark Erebus, and deep Tartarus. Earth, the air and heaven had no existence. Firstly, blackwinged Night laid a germless egg in the bosom of the infinite deeps of Erebus, and from this, after the revolution of long ages, sprang the graceful Eros with his glittering golden wings, swift as the whirlwinds of the tempest. He mated in deep Tartarus with dark Chaos, winged like himself, and thus hatched forth our race, which was the first to see the light.*“

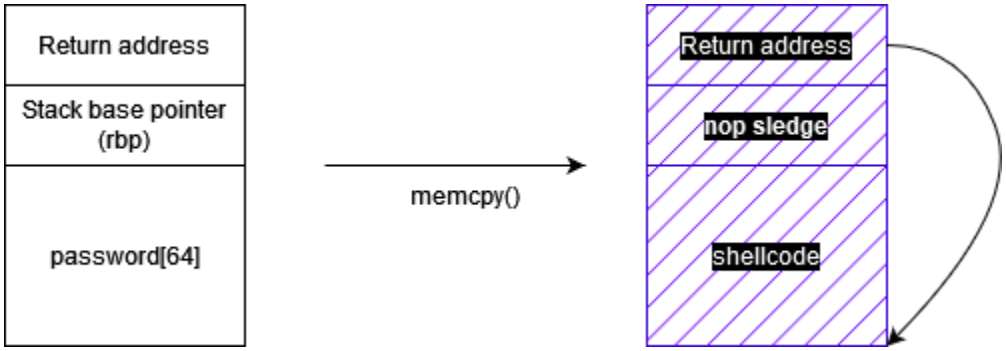
The initial instance of memory corruption was discovered in 1988 where Morris worm found to be exploiting the fingerd Unix application. This particular incident used a buffer overflow vulnerability, that leads to code execution. Subsequently, the awareness of memory corruption spread throughout the security community as a result of the publication of "Smashing the stack for fun and profit" in the 49th edition of Phrack in 1996. Prior to this publication, buffer overflows were acknowledged but not widely attended to. However, since then, researchers, regardless been red or blue, have begun to prioritize this issue. Consequently, memory corruption has wreaked chaos within the security community.

THE FIRST ORDER: The introduction of memory corruptions

Let's look at a traditional case of buffer overflow that will help to understand what makes memory corruption vulnerability and how code execution happens due to memory corruption.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv[])
{
        char password[64];
        strcpy(password, argv[1]);
        if (strcmp(password, "secret") == 0) {
                printf("Sucessfully login\n"); }
        else{
                printf("Password doesn't match. Unanble to login.\n"); }
        return 0;
}
```

The above program presents a clear-cut problem of copying data into a stack buffer without specifying the number of bytes that should be copied. This vulnerability can enable malicious input to execute arbitrary shellcodes by modifying the return address to point it to the shellcode stored in the stack buffer, which will be passed as part of the password buffer. This can be better understood using below illustration:



Similar type of issue can occur in heap as well, where an overflow can cause arbitrary read/write primitives due to control of modification of heap chunk header. Although heap contain more complex issues than the above mentioned like use after free, double free that mostly occurred due to bad implementation of heap allocator.

This series is not about digging into memory safety issues, but here are the common types of cases you will see that are related to memory:

- Uninitialized memory
- Reading/ Writing freed memory (Use after free)
- Illegal read/write or Out of bound read/write
- Null pointer dereferences
- Function pointer modifications
- Memory leaks (usually not treated as security vulnerability)

Besides above there are certain bugs that cause memory corruptions like integer overflows, heap allocator implementation issues etc. Our work is not focusing on these bugs rather on the actual corruptions(main exploitation scenario).

The third order: Categorizing the work

I decided to categorize the mitigations that have been introduced over the years into three general groups:

Let’s quickly try to understand what each categories meant:

First generation mitigations:

These are the techniques that are mostly introduced in early years of protecting memory corruption issues until around 2010. Due to their long-standing presence, they have reached a state of predominantly being stable or deprecated, with no subsequent advancements taking place in these areas.

Second Generation mitigations:

Second generation mitigations are mostly introduced after 2010 and are focused on filling up the gaps that 1st generation mitigations left or for cases where 1st gen mitigations are bypassed. Most of the techniques mentioned here are introduced in different platforms in the last 5 years, hence are still under development or constant improvements.

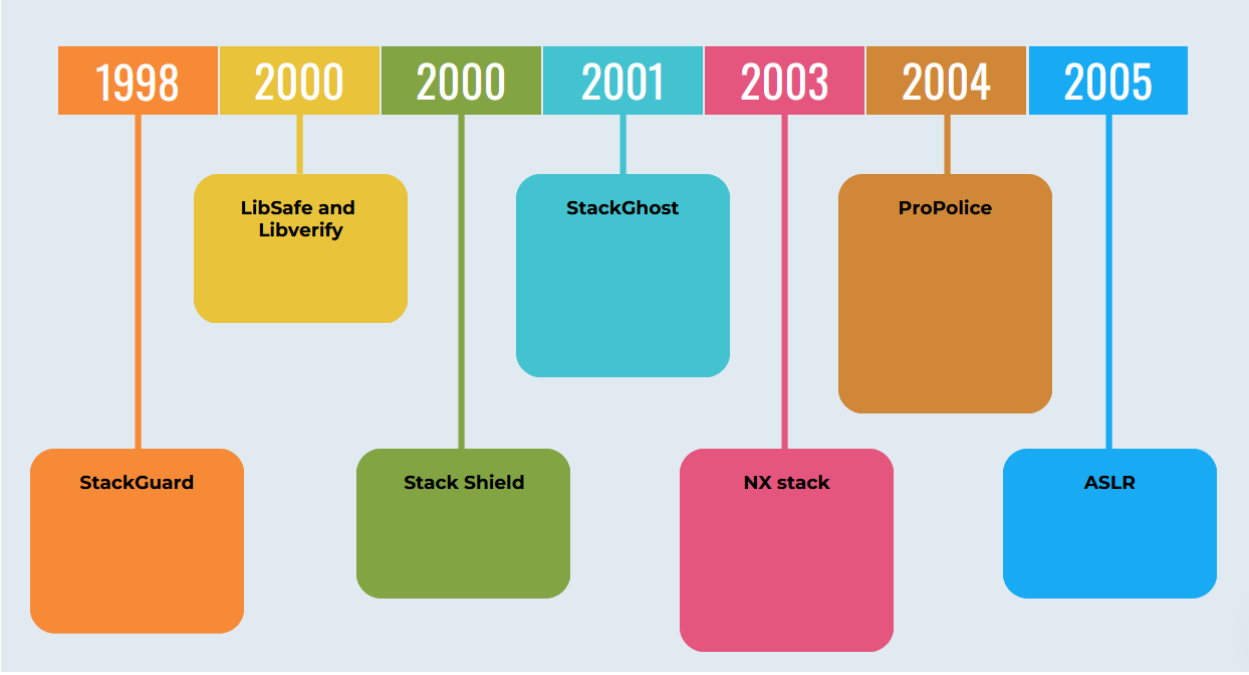
Memory detection tools:

This category is different then above two but serves the common objective of detecting memory errors. This contains mostly tools (techniques in some cases) that are introduced by different vendors to detect memory error conditions during the testing or development stages, and are not intended for deployment in production environments.

Let us examine each part individually in the following three upcoming sections.

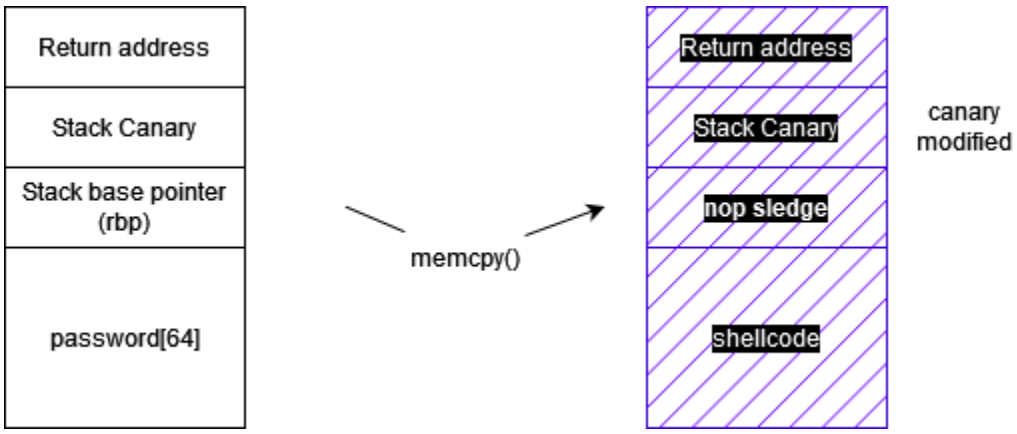
First generation mitigations (The Titans)

The first generation's mitigations are focused on detecting or preventing memory corruptions caused in the program. Over course of years, the timeline of these mitigations looks something like this



Stack Guard

Stack guard, also referred to as Stack smashing protection, was initially introduced in 1997 as a notable defense mechanism against buffer overflow. This innovation was subsequently integrated into gcc 2.7 in the year 1998. The consumer market was first presented with this safeguard in the Immunix distribution during the same year.
The operating principle of stack guard involves adding a random 8 bytes data (called stack canaries) at the starting of the function stack frame. Upon the function's return, a comparison is made to ascertain whether the canary remains unchanged. In the event of an overflow, the canaries are also altered so as to modify the return address. Upon termination, if it is determined that the canary has been tampered, the program gets terminated.



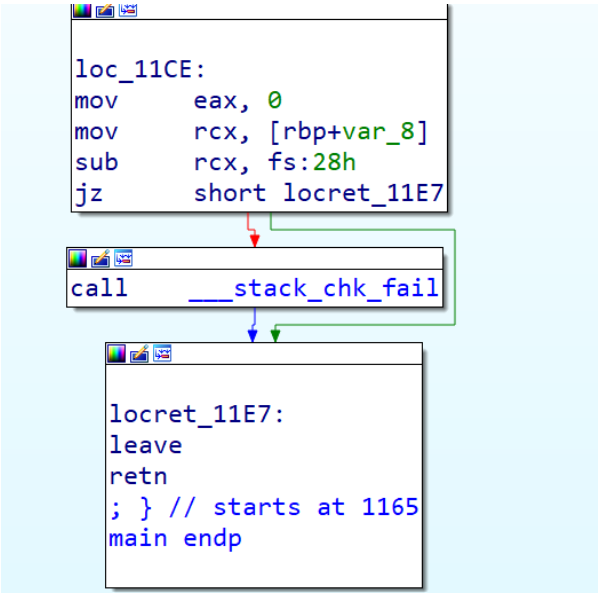
Let's take a look at the stack guard implementation in linux.

Stack canary implementation in Linux:

If you compile the above mentioned classic buffer overflow program in any linux distro with parameter *`fstack-protector`* and check the disassembly in IDA, you will see some canary specific check at starting of main function.

```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_60= qword ptr -60h
var_54= dword ptr -54h
dest= byte ptr -50h
var_8= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 60h
mov     [rbp+var_54], edi
mov     [rbp+var_60], rsi
mov     rax, fs:28h
mov     [rbp+var_8], rax
xor     eax, eax
mov     rax, [rbp+var_60]
add     rax, 8
```

Here *fs:[0x28]* holds the random 8 byte stack guard value that has been saved to stack (showing as *var_8*) as the first thing. At the function epilog you will see the following instructions sequence.



Here the *var_8* (which holds the canary value) is moved to *rcx* and subtracted *fs:0x28* from it to check if it's 0 or not. If it's not zero, implies the canary value has been modified which cause execution of *__stack_chk_fail,* that will terminate the program execution.

> *Note: Use of fs segment in latest x86 linux: The %fs segment register is used to implement the thread pointer. The linear address of the thread pointer is stored at offset 0 relative to the %fs segment register*

On which functions gcc adds stack guards?

From the previously mentioned observations, it can be inferred that the inclusion of stack guard entails the addition of a considerable number of novel instructions to a given function. This, in turn, may result in a slight performance impact. Consequently, compilers such as gcc, or any other compiler for that matter, will only append these stack guard checks to a function if specific prerequisites are satisfied. Thus, the following options are made available within the gcc framework to enable stack guard functionality based on requirement:

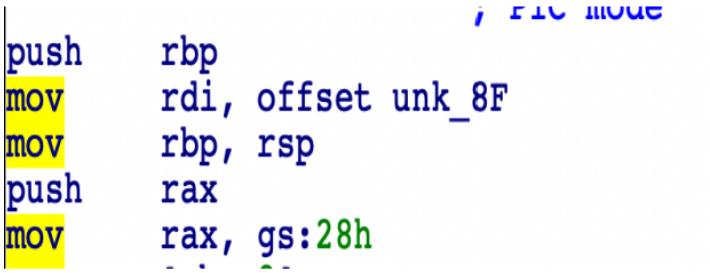
- ***-fstack-protector*** – Adds stackguard on functions that have a local buffer of 8 bytes or more or functions that call *alloca*().
- ***-fstack-protector-strong*** – Adds stack guard on functions that have local array definitions, or have references to local frame addresses.
- ***-fstack-protector-all*** – Adds canary on all functions.

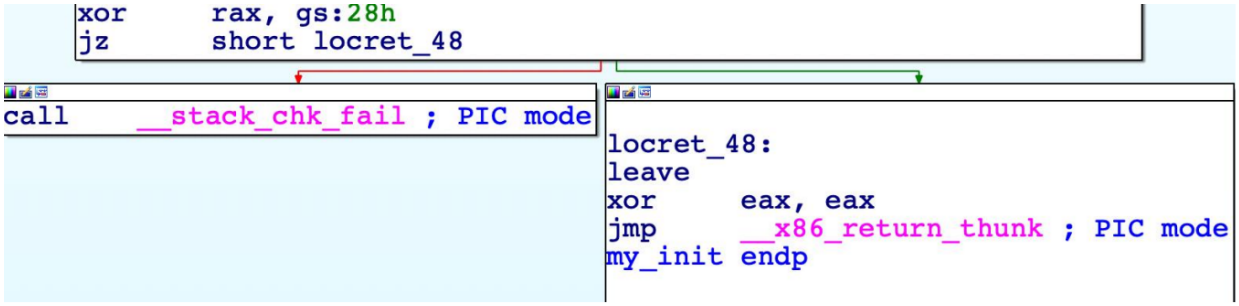
Stack canaries on linux kernel

Stack Canary based protection was introduced in linux kernel 2.6 in 2008. Linux supports following config parameters to define which functions needs canary during build:

- *CONFIG_CC_STACKPROTECTOR* – Similar to *-fstack-protector*. Puts canary at starting of function with stack allocation more than 8 bytes.
- *CONFIG_CC_STACKPROTECTOR_STRONG* – Similar to *-fstack-protector-strong*.
- *CONFIG_CC_STACKPROTECTOR_ALL* - Add canary to all function
- *CONFIG_CC_STACKPROTECTOR_AUTO* – Try to compile the kernel with the best possible option available.
- *CONFIG_CC_STACKPROTECTOR_NONE* – Build kernel without any stack guard protection.

The instruction sequence will looks like below:





Stack Guard in Windows

Stack guard was introduced in windows in 2003 with the visual studio support for */gs* flag.

In case of windows you will going to see following instruction set at the function prolog:

```
mov     rax, cs:__security_cookie
xor     rax, rbp
mov     [rbp+130h+var_18], rax
```

In windows, rather than saving the canary directly like linux implementation, it's first *xored* with base pointer(*rbp*) and then saved in the stack.Extra *xor* operation adds a layer of randomization to canary value since the base pointer itself is random on each program execution due to ASLR. Following will be the code you will find on epilog of function:

```
mov     rcx, [rbp+130h+var_18]
xor     rcx, rbp        ; StackCookie
call    j___security_check_cookie
lea     rsp, [rbp+128h]
pop     rdi
pop     rbp
retn
```

The function call to *j__security_check_cookie* will verify if *rcx* is set to 0 or not. If not, then the function will abort the program.

Implementation in Windows kernel

The Windows kernel also provides support for stack canary based protection. However, upon conducting my analysis, I discovered a limited number of functions that incorporate canary check. Furthermore, there is no explicitly defined criteria known regarding the selection of these functions by the kernel. Nevertheless, you will encounter some instances similar to the ones mentioned below:

prolog:

```
push    rbx
sub     rsp, 70h
mov     rax, qword ptr [ntkrnlmp!__security_cookie (fffff80
xor     rax, rsp
mov     qword ptr [rsp+68h], rax
```

epilog:

```
mov     rcx, qword ptr [rsp+68h]
xor     rcx, rsp
call    ntkrnlmp!__security_check_cookie (fffff8047dc6b0d0)
add     rsp, 70h
pop     Operation (rbx)
ret
```

Note that the kernel doesn't use *gs* or *fs* segments to store random canary but rather has a global variable defined for this.

Limitation of stack canary

The most prominent issue with stack canary's overall architecture is that it's there to detect the overflows rather than protect it. i.e it does not prevent a buffer from overflowing but instead detects the overflow once it has already occurred, during the function return.

Above can be a challenging situation in multiple cases like below:

In kernel space stack guard is not the best solution since the memory mapping in kernel space is linear i.e there is no or very less isolation between different components of kernel. Consequently, if an overflow occurs with the stack guard in place, an attacker can potentially modify other kernel components before the overflow is detected. In summary, if a stack overflow is detected at all on a production system, it is often well after the actual event and after an unknown amount of damage has been done.

Bypassing with brute force:

The success of stack guard heavily depends on the randomization of canary/ guard value. In older or some custom systems, canary values are predefined or pseudo random which makes guessing it easy for the attacker. If the value is guessed correctly, stack canary becomes essentially useless.

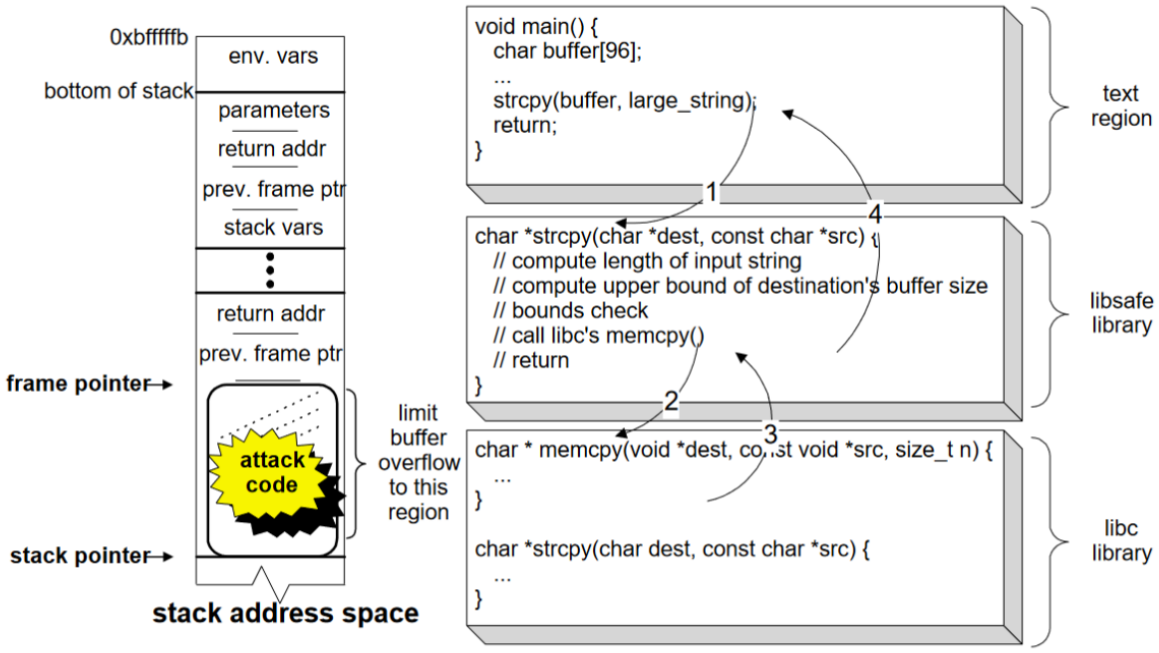
LibSafe and Libverify

LIbsafe was an important idea presented on usenix conference by a few researchers from bell labs and rst corp in 2000. It was targeted for the linux platform and was merged into Debian the same year.

Unlike most other common mitigations, libsafe/libverify can work on pre compiled binary as all of its implementation is present as a dynamic loaded library that can be loaded along with any or all processes.

LibSafe

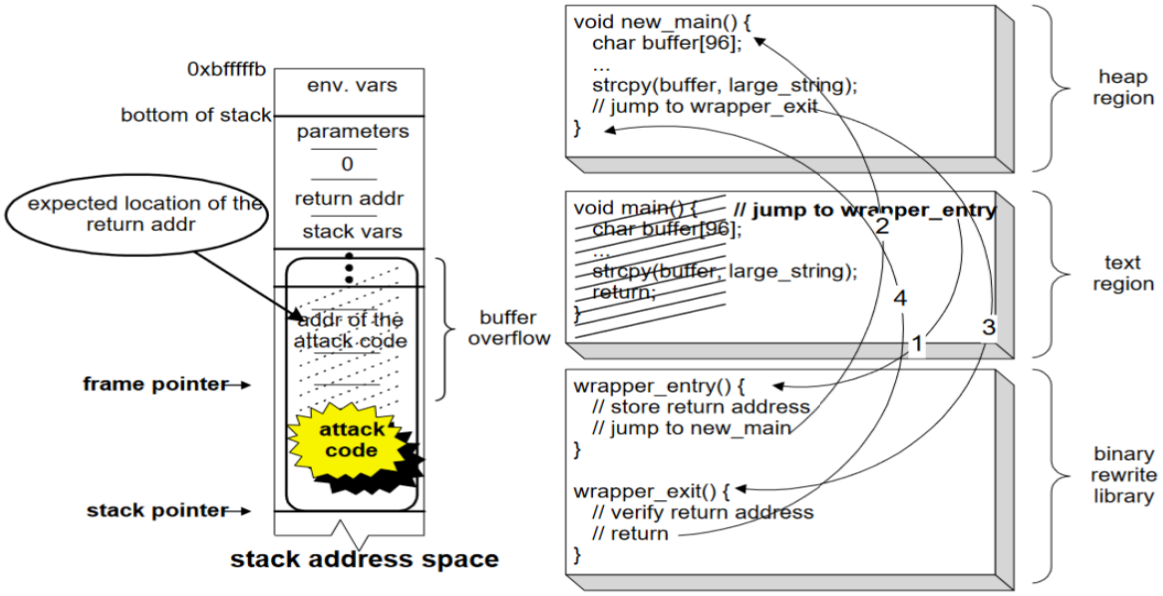
The libsafe intercepts all calls to library functions that are known to be vulnerable from their loaded library. A substitute version of the corresponding function implements the original functionality but in a manner that any buffer overflow are contained within the current stack frame. Detection is based on maximum buffer size that a single write can modify, which get realized by the fact that local buffers cannot be modified beyond the current stack frame. It can be understood more clearly with the below illustration



At the time strcpy() is called, the frame pointer (i.e., the ebp register in the Intel Architecture) will be pointing to a memory location containing the previous frame's frame pointer. Furthermore, the frame pointer separates the stack variables (local to the current function) from the parameters passed to the function. The size of the buffer and all other stack variables residing on the top frame cannot extend beyond the frame pointer—this is a safe upper limit. A correct C program should never explicitly modify any stored frame pointers, nor should it explicitly modify any return addresses (located next to the frame pointers). Libsafe use this knowledge to detect and limit stack buffer overflows. As a result, the attack executed by calling the strcpy() can be detected and terminated before the return address is corrupted.

LibVerify

The libverify library relies on verification of the function's return address before it is used (in a similar way StackGuard is used). It injects the verification code at the start of process execution via rewriting the binary after it is written on the memory. It can be illustrated using the diagram below.



Before the process commences execution, the library is linked with the user code. As part of the linking procedure, the *init()* function in the library is executed. The *init()* function contains a stub to instrument the process such that the canary verification code in the library will be called for all functions in the user code.

The instrumentation includes the following steps:

1. Determine the location and size of the user code.

2. Determine the starting addresses of all functions in the user code.

3. For each function (a) Copy the function to heap memory.(b) Overwrite the first instruction of the original function with a jump to the wrapper entry function. (c) Overwrite the return instruction of the copied function with a jump to the wrapper exit function.

The wrapper entry function saves a copy of the canary value on a canary stack and then jumps to the copied function. The wrapper exit function verifies the current canary value with the canary stack. If the canary value is not found on the canary stack, then the function determines that a buffer overflow has occurred. In contrast to StackGuard, which generates random numbers for use as canaries, libverify uses the actual return address as the canary value for each function. This simplifies the binary instrumentation procedure because no additional data is pushed onto the stack, which means that the relative o sets to all data within each stack frame remain the same.

Resources:

https://www.usenix.org/legacy/publications/library/proceedings/usenix2000/general/full_papers/baratloo/baratloo.pdf

StackShield

Stack shield is an independent toolset that was released in 2000 for linux. It consist of *shieldgcc* and *shieldg++* (replacement of default gcc and g++) to compile c/c++ binary with stackshield protection.

Stack Shield has two main protection methods: the Global Ret Stack (default) and the Ret Range Check. The core feature(Global Ret Stack) of stackshield is to save and verify the return address in a separate memory space named retarray.

It uses two global variables for a function. *rettop* – which stores the end address of retarray and *retptr* which holds the address where next return address need to be saved. On entry to a protected function, the return address is copied from the stack to retarray and *retptr* is incremented. During epilog return addresses saved in stack are not used. Instead of them, the cloned return address stored in retarray are honored. The pseudo assembly for stack shield looks like this:

```
function_prologue:
    pushl   %eax
    pushl   %edx

    movl    retptr,%eax     // retptr is where the clone is saved
    cmpl    %eax,rettop     // if retptr is higher than allowed
    jbe     .LSHIELDPROLOG  // just don't save the clone
    movl    8(%esp),%edx    // get return address from stack
    movl    %edx,(%eax)     // save it in global space
.LSHIELDPROLOG:
    addl    $4,retptr       // always increment retptr

    popl    %edx
    popl    %eax

standard_prologue:
    pushl   %ebp            // saves the frame pointer to stack
    mov     %esp,%ebp       // saves a copy of current %esp
    subl    $108, %esp      // space for local variables

    (function body)

function_epilogue:
    leave                   // copies %ebp into %esp,
                            // and restores %ebp from stack
    pushl    %eax
    pushl    %edx

    addl     $-4,retptr     // allways decrement retptr
    movl     retptr,%eax
    cmpl     %eax,rettop    // is retptr in the reserved memory?
    jbe      .LSHIELDEPILOG // if not, use return address from stack
    movl     (%eax),%edx
    movl     %edx,8(%esp)   // copy clone to stack
```

Stack shield also contain another level of protection where the cloned return address is compared with the one presented in stack, and if they are different, a SYS exit system call is issued, abruptly terminating the program.

Another feature of Stack shield Ret Range Checking which detect and stop attempts to return into addresses higher than that of the variable *shielddatabase*, assumed to mark the base for program's data, where we may say for simplicity, heap and stack are located. The pseudo code for this looks something like this:

```
function_epilogue:
    leave                   // copies %ebp into %esp,
                            // and restores %ebp from stack

    cmpl    $shielddatabase,(%esp)
    jbe     .LSHIELDRETRANGE   // trying to return to a high address?

    movl    $1,%eax
    movl    $-1,%ebx
    int     $0x80           // Abort program execution (SYS_exit)
.LSHIELDRETRANGE:

    ret                     // jump to address on stack's top
```

Resources:

Stackshield homepage: https://www.angelfire.com/sk/stackshield/index.html

More info: https://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/defeat-stackguard.pdf

Stackghost

In 2001, Mike Frantzen from Purdue university came up with a Hardware-facilitated stack overflow protection technique that will be implemented on kernel space, named as StackGhost. In 2001 stackghost was added in OpenBSD but started to be shipped enabled by default in 2004. It was one of the major initial research done against protecting stack overflows, that has inspired many researchers and implementation later in future.. The StackGhost research was mostly targeted for Sun microsystems Sparc architecture but can easily be imported to other architectures.

It uses register windows in SPARC architecture to make stack overflow exploitation harder. Stackghost only needs to be evoked on deep function calls and recursive function calls. From the Wikipedia:

> *It uses a unique hardware feature of the Sun Microsystems SPARC architecture (that being: deferred on-stack in-frame register window spill/fill) to detect modifications of return pointers (a common way for an exploit to hijack execution paths) transparently, automatically protecting all applications without requiring binary or source modifications.*

Along with the above main StackGhost idea, this research also suggested additional mechanisms that can be added with the above method to make stack overflow exploitation harder. These ideas were very raw during that time, hence worth to mention here:

Encoded Return address: Rather than saving the exact return address, a reversible transform can be applied to the actual return address and the result saved process stack. When the return address needs to be accessed, the reverse transform can be applied before the access completes. To retrieve the actual

value a reverse computation is calculated. If the attacker doesn't know the transform or the key to transform, he/she will not be able to redirect the program flow with his own shellcode address.

One of the ways above technique can be used is by using the last two LSB in address since they are always zero due to 32 bit word alignment required for each instruction in SPARC architecture. The transformation can invert one or both least significant bits. A corrupted return address will be detected when these bits are not set during inverse transformation.

Return Address stack: A corrupt return pointer can also be detected by keeping a return-address stack. Every time a return pointer is saved to the program stack, the handler can keep another copy in memory not accessible to the userland process (a return-address stack). A corrupt return pointer is detected if a function tries to return to a location not saved in the return stack.

The idea is to have a return address stack as a FIFO queue. A refined approach to designing a return-address stack is to add a small hash table in the PCB. Every time a register window needs to be cleansed, the mechanism would add an entry into the hash table (indexed off the base address of the stack frame). And then store the base address to use as the comparison tag, the return pointer, and a random 32- bit number. In the place of the return address in the stack frame, it would place a copy of the random number. When StackGhost retrieves the stack frame to refill the register window, it can compare the random number on the stack with its image in the hash table. If the instances do not match, an exploit has occurred and the program must be aborted.

XOR Cookie – Another important mention in Stackghost paper is to protect the return address by XORing a fixed cookie with the return address. XORing the cookie before it is saved and xoring again after it popped off preserve the legitimate pointer but distort the attack. Stackghost's idea was to build either a 13 bit preset cookie in OpenBSD kernel that will remain same throughout the system/between processes or have a per process cookie.

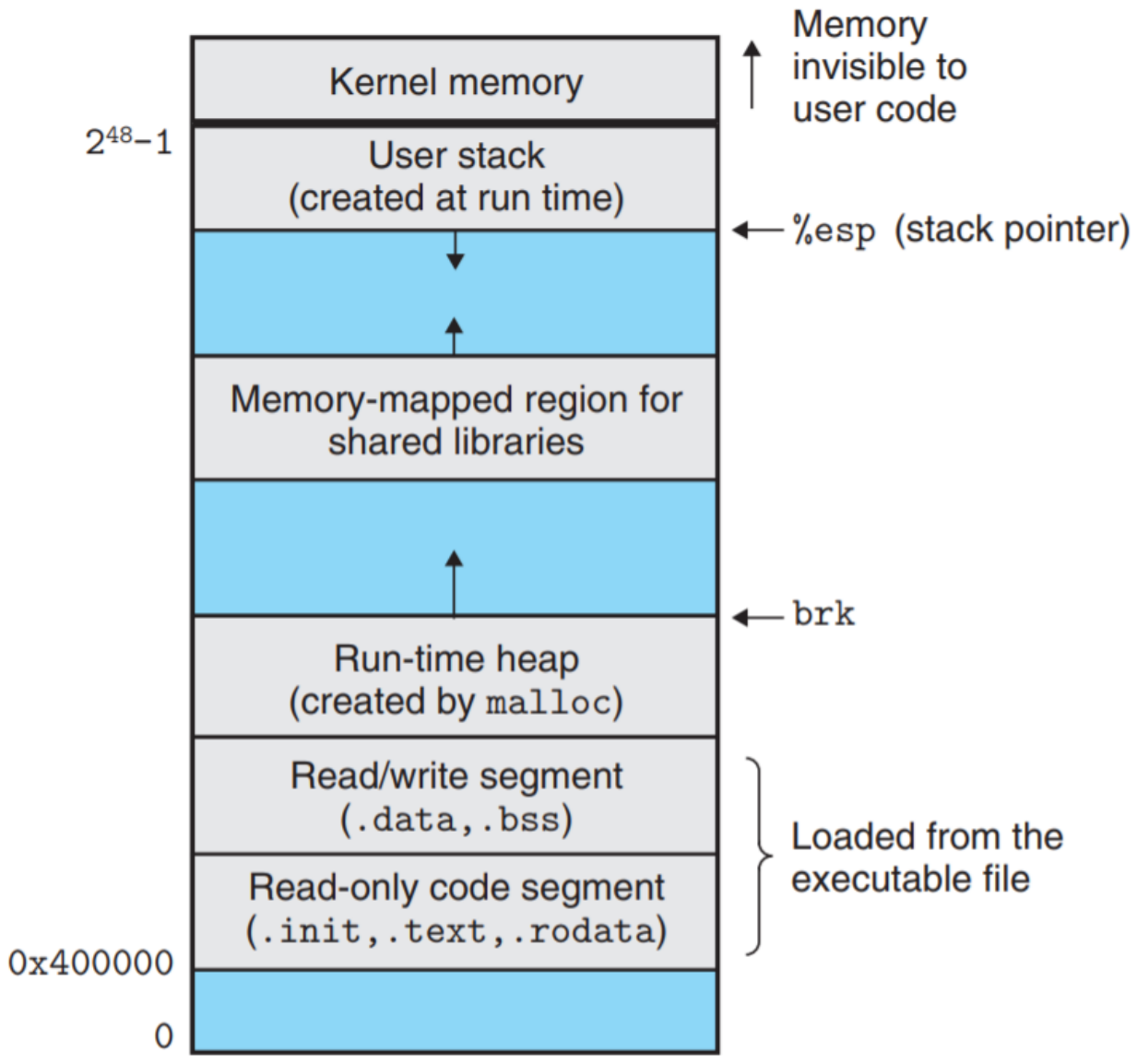
Encrypted Stack Frame – Corrupted return can also be detected by encrypting part of the stack frame when the window is written to the stack and decrypting it during retrieval. Although it is going to have a significant performance impact.

Limitations of Stack Ghost

- Randomness of XOR cookie is low, making it easily predicted and align with the actual address.
- Most of the techniques that stackghost use are based on detection but not prevention. An exploit can still cause Denial of service.
- StackGhost will not stop every exploit, nor will it guarantee security. Exploits that StackGhost will not stop include:
    - 1. A corrupted function pointer (atexit table, .dtors, etc.)
    - 2. Data corruption leading to further insecure conditions.
    - 3. "Somehow" overwriting a frame pointer with a frame pointer from far shallower in the calling sequence. It will short circuit backwards through a functions' callers and may skip vital security checks.

Execution space protection

A process or kernel memory is partitioned into several segments that serve different purposes. For example, a process memory typically consists of a executable section (.text), read only data (.rodata), read/writable data (.data), a stack and a heap. Below is one way of memory layout



Since each section has different usage, they required different memory permissions. For instance, by default, the .rodata section should exclusively possess read permissions, while the .text section should possess both read and execute permissions.

It is imperative for operating systems to effectively enforce these permissions in order to circumvent unexpected behavior or misuse of memory. For instance, there may exist scenarios wherein an attacker gains the privilege to write to any memory location within a process. This privilege can be exploited by

modifying the program's behavior through altering the instructions within the text section of the process's memory. However, with accurately enforced memory permissions, any attempts to write to the text section of the process's memory will be thwarted.

> Note: You can check the memory permissions of each section for a process in linux by using *cat /proc/<pid>/maps*. Similarly the process memory permissions can be checked in windbg attached to the target process using *!address* command in windbg attached to the target process.

As a part of ESP, your loader will mark the memory region as Non executable to block execution of any shellcode in that area. In a perfectly implemented operating system, following memory regions are marked as non executable:

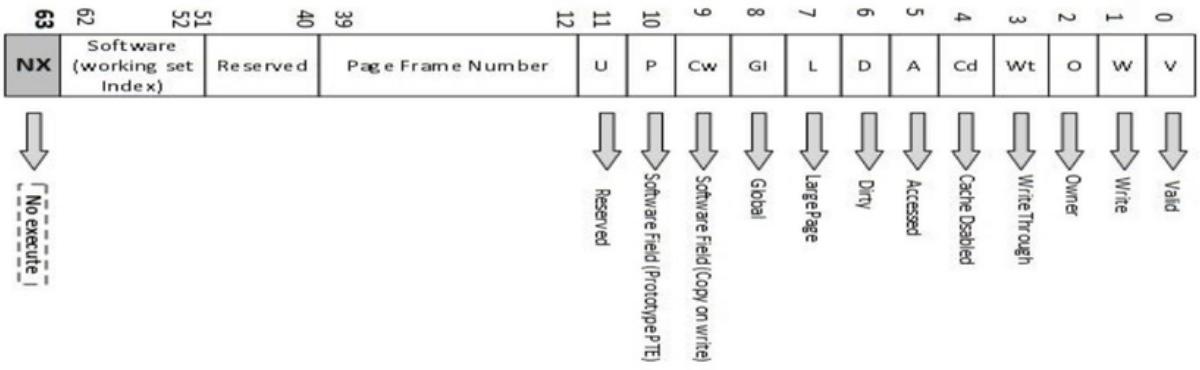
- All non writable sections including .data, .rodata, .bss etc
- Stack and heap of process

The above implementation will prevent any exploitation of process address space against any arbitrary shellcode execution present in these memory regions. To make this execution prevention more readily implementable, chipmakers have implemented this as a hardware feature called as Data Execution prevention/NX stack explained below:

NX protection

In 1998, a team name “solar designers” developed a patch for linux that will make the stack non executable. But due to the gcc’s limitations of using stack to store trampoline function and executing it, the implementation didn't get merged in gcc until 2004 (which was after the introduction of NX bit in x86 processors).

Intel and AMD introduced NX bit support in 2001 in AMD64 and Intel Itanium processors respectively. This is an extra bit added in the page table entry to mark the page as non executable.



In 64 bit machine (as mentioned in above image) the bit 63rd (MSB in address) will be used to mark a page as non executable. This will prevent the impact of buffer overflow exploitation where attacker try to execute shellcode present in the user input buffer in stack or heap.

This feature was introduced in windows with the name DEP (Data execution prevention) in 2003 in windows XP. Currently, NX protection is available and used at all major operating systems at userspace and kernel space both.

Limitations of NX stack

Clearly the protection will only protect against the case where an attacker will try to pass the shellcode in user buffer and execute it later. The return address redirection is still possible. Attackers can still redirect the program flow to a function already present inside process memory. One such famous attack in linux is “return to libc”. This method is also called ROP or JOP (where ROP stands for Return oriented programming and JOP for Jump oriented programming).

Resources:

https://academickids.com/encyclopedia/index.php/NX_bit

Propolice

Propolice is the stack smashing protection patches that are added by IBM in gcc in year 2004. The idea was based on improving the coverage and detection of existing Stack Guard protection.

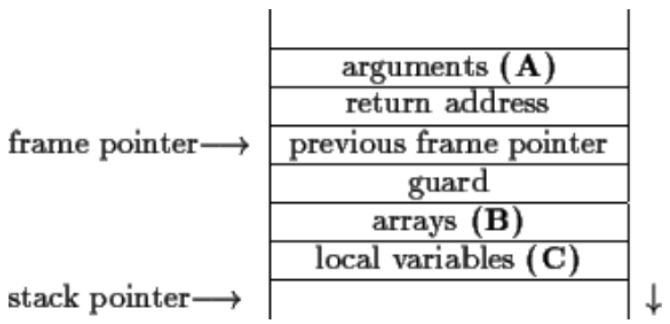
The improvement in propolice method compared to stackguard is the location of the guard and the protection of function pointers. The guard is inserted next to the previous frame pointer and it is prior to an array, which is the location where an attack can begin to destroy the stack.

The Propolice extension patch consists of following major changes to existing gcc:

- the reordering of local variables to place buffers after pointers to avoid the corruption of pointers that could be used to further corrupt arbitrary memory locations,
- the copying of pointers in function arguments to an area preceding local variable buffers to prevent the corruption of pointers that could be used to further corrupt arbitrary memory locations, and the
- omission of instrumentation code from some functions to decrease the performance overhead.

Safety function model:

To decrease the impact of stack overflow, propolice includes an improved function stack model compared to stackguard, that can be followed to minimize the impact of stack overflow.



safety function model, which involves a limitation of stack usage in the following manner:

- the location (A) has no array or pointer variable
- the location (B) has arrays or structures that contains an array
- the location (C) has no array

This model has the following properties:

- The memory locations outside of a function frame cannot be damaged when the function returns.
    - The location (B) is the only vulnerable location where an attack can begin to destroy the stack. Damage outside of the function frame can be detected by the verification of the guard value. If damage occurs outside of the frame, the program execution stops.
- An attack on pointer variables outside of a function frame will not succeed. The attack could only succeed if the following conditions were satisfied: (1) the attacker changes the value of the function pointer, and (2) he calls a function using the function pointer. In order to achieve the second condition, the function pointer must be visible from the function, but our assumption says this information is beyond the function scope. Therefore, the second condition can't be satisfied, and the attack will always fail.
- An attack on pointer variables in a function frame will not succeed. The location (B) is the only vulnerable location for a stack-smashing attack, and the damage goes away from area (C). Therefore, the area (C) is safe from the attack.

Pointer protection

Another common stack overflow exploitation scenario is when the vulnerable function consists of a function pointer as local variable like below:

```
void bar( void (*func1)() )
{
        void (*func2)();
        char buf[128];
        .......
        strcpy (buf, getenv ("HOME"));
        (*func1)(); (*func2)();
}
```

In order to protect function pointers from stack-smashing attacks, propolice change the stack location of each variables to be consistent with the safe function model. It makes a new local variable, copying the argument ``func1'' to it, and changing the reference to ``func1''to use the new local variable.

```
void bar( void (*tmpfunc1)() )
{
        char buf[128];
        void (*func2)();
        void (*func1)(); func1 = tmpfunc1;
        .......
        strcpy (buf, getenv ("HOME"));
        (*func1)(); (*func2)();
}
```

The propolice methodology is still used by gcc and other compilers to a greater extent.

Resources

https://web.archive.org/web/20040603202721/http://www.research.ibm.com/trl/projects/security/ssp/

https://dominoweb.draco.res.ibm.com/reports/rt0371.pdf

Honorable mention:

In windows 10, Microsoft has added a new mitigation named Arbitrary code guard in Windows kernel, which is based on setting and verifying memory permissions for specific pages based on certain policy and blocking the attempt that breaks the policy. ACG is based on prevents a process from doing these two things:

- Allocating new executable memory (without an image file backing it)
- Modifying any existing executable memory by writing to it.

You can read more about ACG here:
https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution/

ASLR (Address Space Layout Randomization)

The idea of ASLR was first introduced in PAX project in 2001. Later it was added in OpenBSD in 2003, followed by linux in 2005 and Windows vista in 2007. The main goal of ASLR is to prevent the exploitation due to memory corruption rather than the detecting/preventing corruption itself.

The exploitation of a memory corruption vulnerability entails the act of redirecting the return address to an alternative function or shellcode address that is either malicious or unexpected. In order for the exploitation to be successful, the attacker must possess knowledge of the specific target address that they intend to execute. Drawing inspiration from this, Address Space Layout Randomization (ASLR) was implemented as a means to randomize the addresses of most, if not all, sections within a process's

memory. By doing so, this serves to prevent attackers from being able to predict the location of any malicious function or shellcode addresses.

ASLR in action

ASLR in linux

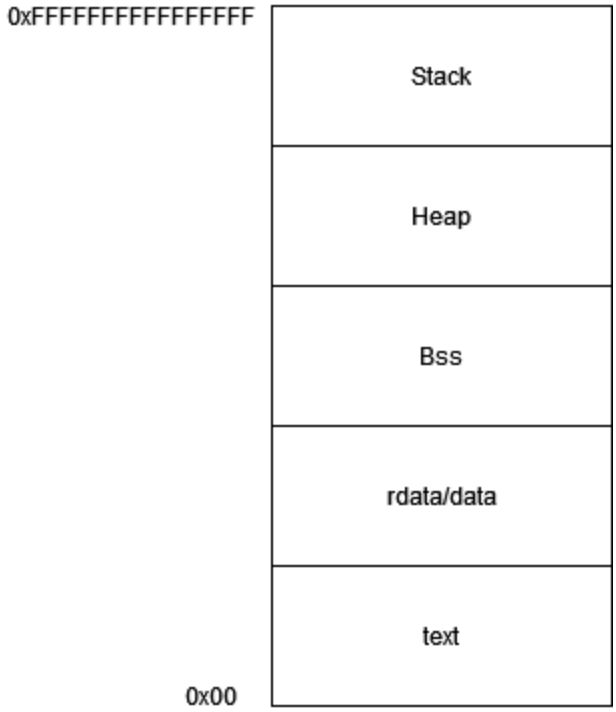
Let's take a look at simple c program below to understand the impact of ASLR:

```c
#include <stdio.h>

int bss_var;

int main()
{
        int stack_var = 10;
        int heap_var = malloc(120);
        long int text_var = &main;
        char *a = "Hello";
        printf("Address of variable in stack is 0x%lx\n",&stack_var);
        printf("Address of variable in heap is 0x%lx\n",heap_var);
        printf("Address of variable in rdata is 0x%lx\n",a);
        printf("Address of variable in bss is 0x%lx\n", &bss_var);
        printf("Address of variable in text is 0x%lx\n", text_var);
        return 0;
}
```

When you execute the below code multiple times in linux, you will get the output like below:

```
shubham@MININT-1T2PIDD:~/memory_protection$ ./a.out
Address of variable in stack is 0x7fffc3c31f44
Address of variable in heap is 0x55629504e2a0
Address of variable in rdata is 0x556293764008
Address of variable in bss is 0x55629376603c
Address of variable in text is 0x556293763145
shubham@MININT-1T2PIDD:~/memory_protection$ ./a.out
Address of variable in stack is 0x7ffdb2f31a24
Address of variable in heap is 0x560c588042a0
Address of variable in rdata is 0x560c56aea008
Address of variable in bss is 0x560c56aec03c
Address of variable in text is 0x560c56ae9145
shubham@MININT-1T2PIDD:~/memory_protection$ ./a.out
Address of variable in stack is 0x7ffe2005a5c4
Address of variable in heap is 0x557fa68e72a0
Address of variable in rdata is 0x557fa5258008
Address of variable in bss is 0x557fa525a03c
Address of variable in text is 0x557fa5257145
shubham@MININT-1T2PIDD:~/memory_protection$ ./a.out
Address of variable in stack is 0x7ffe25223104
Address of variable in heap is 0x5580eaaae2a0
Address of variable in rdata is 0x5580e9c05008
Address of variable in bss is 0x5580e9c0703c
Address of variable in text is 0x5580e9c04145
shubham@MININT-1T2PIDD:~/memory_protection$ ./a.out
Address of variable in stack is 0x7fff40b10764
Address of variable in heap is 0x55d0a2fc42a0
Address of variable in rdata is 0x55d0a2a2c008
Address of variable in bss is 0x55d0a2a2e03c
Address of variable in text is 0x55d0a2a2b145
```

From the above output, we can conclude that linux have following memory layout:



Although the order of the various sections remains consistent, the different addresses of variables during each execution imply that the base address for each section changes. However, it is worth noting that the text section possesses a fixed address. Due to this element of randomization, accurately determining the location of certain data or shellcode in the process memory becomes exceedingly challenging, thereby making it difficult for an attacker to exploit the buffer overflow scenario for the purpose of executing a shellcode.

ASLR in linux kernel (KASLR)

Linux kernel added support for ASLR in 2006 (v2.6) which known as KASLR – Kernel address space layout randomization. KASLR patches has introduced following changes in kernel memory:

- Kernel image which used to reside at a static address in lowmem can now be present at any memory location in kernel memory.
- Initial loading addresses of modules are randomized.
- Both virtual and physical addresses of components get randomized.
- Other memory regions like *vmalloc*, *vmap* and *ioremap* area are also randomized.
- Later the linux kernel also introduced the *KSTACK_RANDOMIZE* feature that will randomize the kernel stack base created on each syscall.

Let’s verify the impact of KASLR in linux kernel:

You need to load the below compiled driver to understand the effect of KASLR in linux kernel:

```c
#include <linux/module.h>
#include <linux/slab.h>
int kernel_bss;

int kernel_rodata = 10;

static int my_init(void)
{
    char *buf;
    int stack_var = 10;
    buf = kmalloc(200, GFP_ATOMIC);
    printk(KERN_INFO "Module loaded.\n");
    printk(KERN_INFO "bss section address 0x%lx!\n", &kernel_bss);
    printk(KERN_INFO "rodata section address 0x%lx!\n", &kernel_rodata);
    printk(KERN_INFO "Stack variable address is 0x%lx!\n", &stack_var);
    printk(KERN_INFO "text section address 0x%lx!\n", &my_init);
    printk(KERN_INFO "vmalloc slab address 0x%lx!\n", buf);
    return  0;
}

static void my_exit(void)
{
    printk(KERN_INFO "Module unloaded.\n");

    return;
}

module_init(my_init);
module_exit(my_exit);
```

Below is the *dmesg* output:

```
[ 3229.388978] Module loaded.
[ 3229.388982] bss section address 0xffffffffc098e400!
[ 3229.388984] rodata section address 0xffffffffc098e000!
[ 3229.388985] Stack variable address is 0xffffb13fc365fde4!
[ 3229.388986] text section address 0xffffffffc098c000!
[ 3229.388987] vmalloc slab address 0xffff8de172f92e00!
[ 3253.579739] Module unloaded.
[ 3257.392020] Module loaded.
[ 3257.392025] bss section address 0xffffffffc098e400!
[ 3257.392026] rodata section address 0xffffffffc098e000!
[ 3257.392028] Stack variable address is 0xffffb13fc25b7de4!
[ 3257.392029] text section address 0xffffffffc098c000!
[ 3257.392030] vmalloc slab address 0xffff8de146ff9100!
[ 3259.044467] Module unloaded.
[ 3261.960543] Module loaded.
[ 3261.960547] bss section address 0xffffffffc098e400!
[ 3261.960549] rodata section address 0xffffffffc098e000!
[ 3261.960550] Stack variable address is 0xffffb13fc25ebde4!
[ 3261.960551] text section address 0xffffffffc098c000!
[ 3261.960552] vmalloc slab address 0xffff8de146ff9700!
[ 3263.783888] Module unloaded.
[ 3265.379130] Module loaded.
[ 3265.379134] bss section address 0xffffffffc098e400!
[ 3265.379135] rodata section address 0xffffffffc098e000!
[ 3265.379137] Stack variable address is 0xffffb13fc3627de4!
[ 3265.379138] text section address 0xffffffffc098c000!
[ 3265.379139] vmalloc slab address 0xffff8de172ffb500!
[ 3266.611434] Module unloaded.
```

You will notice that only the stack and heap is randomized. Other sections like *.rodata* and *.text* are not randomized since the module is loaded unloaded frequently, hence getting the same address each loading. But on reboot, you will notice the different address assign to those as well.

```
[   30.133563] Module loaded.
[   30.133564] bss section address 0xffffffffc0952400!
[   30.133564] rodata section address 0xffffffffc0952000!
[   30.133565] Stack variable address is 0xffff9eb9808cfde4!
[   30.133565] text section address 0xffffffffc0950000!
[   30.133565] vmalloc slab address 0xffff907741a1c300!
```

Other observation you will notice for KASLR is that the randomization of address is very less. Only the base addresses of sections are changing but the offsets are pretty much the same.

ASLR in windows

Let's compile the above program in Visual Studio 2022 in windows 11 and verify the output:

```
C:\Users\shubhamdubey\source\repos\aslr_testing\x64\Debug>aslr_testing.exe
Address of variable in stack is 0x828ff9c4
Address of variable in heap is 0x6e7ec550
Address of variable in rdata is 0x775b9ca4
Address of variable in bss is 0x775bc8cc
Address of variable in text is 0x775b1267

C:\Users\shubhamdubey\source\repos\aslr_testing\x64\Debug>aslr_testing.exe
Address of variable in stack is 0xef38f814
Address of variable in heap is 0x48a9b9a0
Address of variable in rdata is 0x775b9ca4
Address of variable in bss is 0x775bc8cc
Address of variable in text is 0x775b1267

C:\Users\shubhamdubey\source\repos\aslr_testing\x64\Debug>aslr_testing.exe
Address of variable in stack is 0x78ff8d4
Address of variable in heap is 0x6548adb0
Address of variable in rdata is 0x775b9ca4
Address of variable in bss is 0x775bc8cc
Address of variable in text is 0x775b1267

C:\Users\shubhamdubey\source\repos\aslr_testing\x64\Debug>aslr_testing.exe
Address of variable in stack is 0x590ff7f4
Address of variable in heap is 0x5a2cadb0
Address of variable in rdata is 0x775b9ca4
Address of variable in bss is 0x775bc8cc
Address of variable in text is 0x775b1267
```

One may observe that, in addition to the heap and stack, all other variables are located at a fixed address. This suggests that the design of Windows Address Space Layout Randomization (ASLR) takes into consideration the fact that the majority of buffer overflow exploits involve shellcode in the heap and stack, thus necessitating randomization. In order to minimize the overhead of randomization, it is not applied to the remaining sections of the process memory.

We will skip the part of verifying ASLR in windows kernel for now but it can be easily looked with windbg !address extension used with kernel debugger attached.

Limitation of ASLR

Address brute force due to Low entropy

The quality of mitigation provided by ASLR depends on the randomization of the different address spaces in process memory. ASLR is implemented in almost all types of operating systems but has different levels of randomization for different address spaces. Even from the above ASLR POCs, you will notice that the randomization level in windows is quite low compared to linux. Due to less randomization, it became very easy for attackers to guess the address where shellcode or gadgets reside. Moreover, most ASLR implementations have only a few bytes to randomize and keep base address or certain MSB bytes same.

You can easily look for lots of writeups and project to bypass ASLR by bruteforcing the address.

Having a module loaded with no ASLR support

In the Linux operating system, the binaries are constructed by default with the *-fPIC* compiler option in gcc. This particular option indicates that the resulting code will possess the necessary functionalities to enable relocation. This is an essential prerequisite for Address Space Layout Randomization (ASLR) to be operational within the binaries' address space. In the event that the binary is not compiled with this option, or if it is explicitly stated that the binary or library does not support relocation, different addresses of different sections at which the binary or library is loaded into memory will remain consistent on each execution.

Sometimes when a binary itself is fully ASLR compatible, it may have a module/dll loaded that doesn't support the randomization. In those cases, an attacker can try to find ROP gadgets in that unsupported ASLR binary for successful shellcode execution.

Memory address leak

The primary adversary encountered by any ASLR implementation is the exposure of addresses to the user. It can be understood using below case:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
    int secret = 42;
    char username[100];
    char buffer[100];
    // Prompt the user for input
    printf("Enter a string: ");
    fflush(stdout);
    fgets(buffer, 100, stdin);
    // Use the format string vulnerability to leak the address of the
username variable
    printf(buffer);
    scanf("Enter username: %d\n", &username);
    return 0;
}
```

In the above program, there exists an overflow in the *username* variable, which provides attackers with the opportunity to inject shellcode into the username data and subsequently execute it. However, the presence of Address Space Layout Randomization (ASLR) should ideally make it difficult to guess the address of the shellcode. Nevertheless, the program also exhibits a format string vulnerability due to the utilization of *printf(buffer)*, which, if correctly formatted (e.g., by passing "*%p%p%p..*." as the input on the first prompt), will disclose the address of the *username* buffer. Upon receiving these leaked addresses, an attacker can modify the payload in the second prompt in order to successfully redirect the return address to the shellcode. This entire configuration renders the randomization of the stack meaningless.

On similar ground, Address leak is a major culprit of KASLR bypasses in kernel space in windows and linux.

Heap spraying to bypass ASLR

The idea behind this technique is to make multiple addresses lead to the shellcode by filling the memory of the application with lots of copies of it, which will lead to its execution with a very high probability. The main problem here is guaranteeing that these addresses point to the start of it and not to the middle. This can be achieved by having a huge amount of nop bytes (called NOP slide, NOP sled, or NOP ramp), or any instructions that don't have any major effect, such as *xor ecx, ecx*:

During exploitation of a security issue, the application code can often be made to read an address from an arbitrary location in memory. This address is then used by the code as the address of a function to execute. If the exploit can force the application to read this address from the sprayed heap, it can control the flow of execution when the code uses that address as a function pointer and redirects it to the sprayed heap. If the exploit succeeds in redirecting control flow to the sprayed heap, the bytes there will be executed, allowing the exploit to perform whatever actions the attacker wants. The whole concept will work even on the presence of ASLR.

Using Sidechannel to break ASLR

There are multiple researchers published since 2016 that focus on bypassing ASLR or guessing target address space using sidechannel attacks. One such research named “Jump over ASLR” published in 2016, uses BTB buffer to bypass ASLR randomization in userspace or kernel space. Below is few key points from the research.

This new attack can recover all random bits of the kernel addresses and reduce the entropy of user-level randomization by using side-channel information from the Branch Target Buffer (BTB). The key insight that makes the new BTB-based side-channel possible is that the BTB collisions between two user-level processes, and between a user process and the kernel, can be created by the attacker in a controlled and robust manner. The collisions can be easily detected by the attacker because they impact the timing of the attacker-controlled code. Identifying the BTB collisions allows the attacker to determine the exact locations of known branch instructions in the code segment of the kernel or of the victim process, thus disclosing the ASLR offset.

The attacks exploit two types of collisions in the BTB. The first collision type, exploited to bypass KASLR, is between a user-level branch and a kernel-level branch - they call it cross- domain collisions, or CDC. CDC occurs because these two branches, located at different virtual addresses, can map to the same entry in the BTB with the same target address. The reason is that the BTB addressing schemes in recent processors ignore the upper-order bits of the address, thus trading off some performance for lower design complexity. The second type of BTB collisions is between two user-level branches that belong to two different applications. They call these collisions same-domain collisions, or SDC. SDCs are used to attack user-level ASLR, allowing one process to identify the ASLR offset used in another. An SDC occurs when two branches, one in each process, have the same virtual address and the same target.

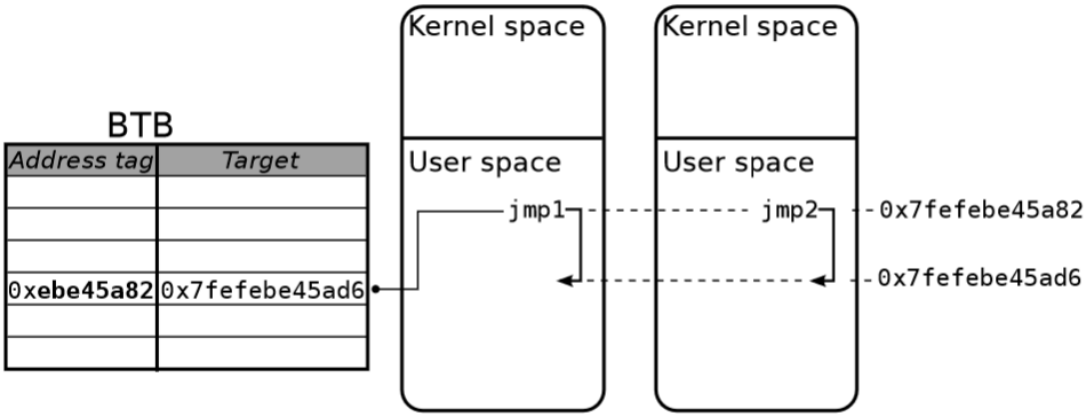


Fig. 1: SDC Example

You can read more detail about the research here: https://www.cs.ucr.edu/~nael/pubs/micro16.pdf

Blindside Attack

Blindside attack discovered by few researchers from Stevens Institute of Technology in New Jersey, ETH Zurich, and the Vrije University in Amsterdam. in 2020 (2 years after Specter, Meltdown surfaced) used Speculative execution modern processor feature to Bypass KASLR. The goal of blindside research was to make KASLR brute forcing (probing) more significant and faster.

When performing probing to bypass KASLR, the non crash resistance nature of linux kernel makes KASLR probing mere a concept. Using speculative execution processor feature for crash suppression allows the elevation of basic memory write vulnerabilities into powerful speculative probing primitives that leak through microarchitectural side effects. Such primitives can repeatedly probe victim memory and break strong randomization schemes without crashes and bypass all deployed mitigations against Spectre like attacks. The key idea behind speculative probing is to break Spectre mitigations using memory corruption and resurrect Spectre style disclosure primitives to mount practical blind software exploits. This became possible since crashes and the probe execution in general are suppressed on speculative paths. You can read about the research here https://download.vusec.net/papers/blindside_ccs20.pdf

KASLR information leak mitigation – kptr_restrict and dmesg_restrict

Kaslr was implemented into the Linux kernel in the year 2006. Since its implementation, various methods have been developed to circumvent kaslr, thereby introducing vulnerabilities into the Linux kernel. These bypasses primarily occur due to the presence of information leaks. These leaks can manifest as either pointer leaks, such as leaks of pointers to structs or heap/stack areas, or content leaks. Illustrative examples of associated CVEs include CVE-2019-10639 (Remote kernel pointer leak), CVE-2017- 14954.

To minimize the impact of address leak linux kernel has added features *kptr_restrict* and *dmesg_restrict*.

ptr_restrict - This indicates whether restrictions are placed on exposing kernel addresses via /proc and other interfaces. When *kptr_restrict* is set to (1), kernel pointers printed using the *%pK* format specifier will be replaced with 0's unless the user has *CAP_SYSLOG*.

dmesg_restrict - This indicates whether unprivileged users are prevented from using dmesg to view messages from the kernel's log buffer. When dmesg_restrict is set, users must have *CAP_SYSLOG* to use dmesg.

The FGKASLR – Fine gained KASLR

Added in linux kernel in 2020, FGKASLR is a replacement of existing KASLR in linux to improve KASLR capability against code reuse attack. Here are few details about FGKASLR from their commit detail:

KASLR was merged into the kernel with the objective of increasing the difficulty of code reuse attacks. Code reuse attacks reused existing code snippets to get around existing memory protections. They exploit software bugs which expose addresses of useful code snippets to control the flow of execution for their own nefarious purposes. KASLR moves the entire kernel code text as a unit at boot time in order to make addresses less predictable. The order of the code within the segment is unchanged - only the base address is shifted. There are a few shortcomings to this algorithm.

1. Low Entropy - there are only so many locations the kernel can fit in. This means an attacker could guess without too much trouble.

2. Knowledge of a single address can reveal the offset of the base address, exposing all other locations for a published/known kernel image.

3. Info leaks abound.

Finer grained ASLR has been proposed as a way to make ASLR more resistant to info leaks. It is not a new concept at all, and there are many variations possible. Function reordering is an implementation of finer grained ASLR which randomizes the layout of an address space on a function level granularity.

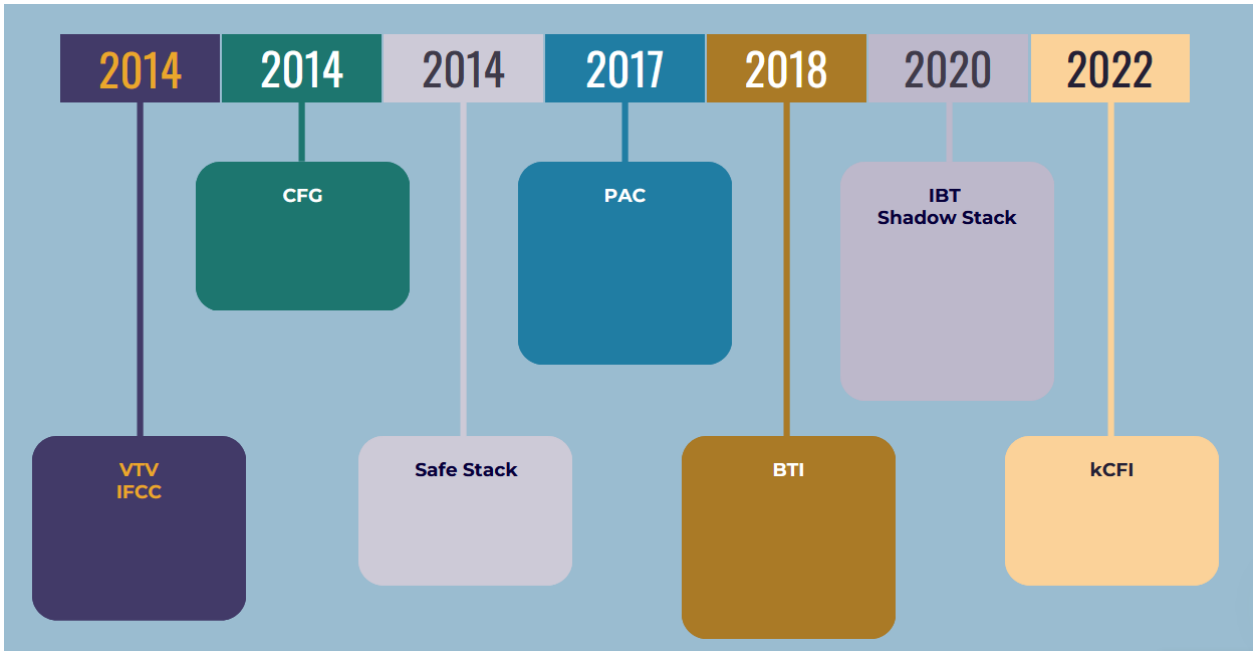
Second generation mitigations (Gods)

After the introduction of ASLR, the advancement in mitigations against memory corruption has been impeded due to the implementation of sufficient mitigations across all platforms in the early 2000s. However, over time, offensive researchers have continuously presented bypasses and limitations of these initial generation mitigation techniques. Some of the primary limitations that persist include

- the presence of ROP, JOP, or any form of code reuse attacks, even with the incorporation of most of the aforementioned techniques.
- indirect function pointers remain unsecured
- heap-based memory corruption can still result in code execution.

These challenges have prompted the security industry to shift its focus towards mitigations that play a crucial role in preventing the exploitation of memory corruption, rather than solely detecting or preventing the corruption itself. This is particularly important in scenarios where the first generation techniques fail, whether due to the aforementioned issues or any other reasons.

In response to these requirements, the security industry has primarily concentrated on the concept of CFI (Control Flow Integrity). CFI serves as the fundamental principle behind all second generation techniques, which will be discussed below.

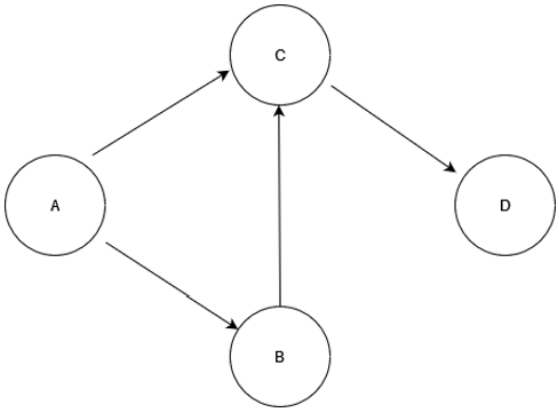


Control Flow Integrity (CFI)

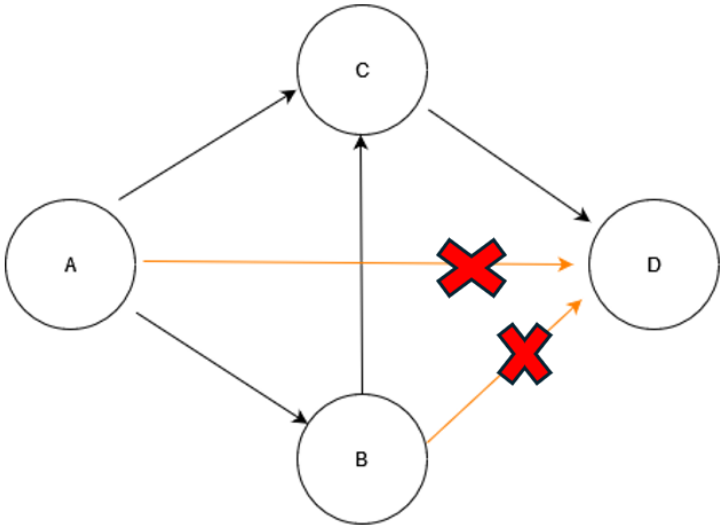
Control flow integrity based mitigation techniques are the one that mitigate the exploitation of memory corruption against arbitrary code execution that can happen due to cases like function pointer modification or virtual table pointer modification. Note that they don't protect or detect the memory corruption itself rather than prevent its exploitation. From the name you can guess that it work to maintain the control flow integrity of program execution with one goal to prevent or detect any illegal branches or redirections. From the CFI wiki:

> Attackers seek to inject code into a program to make use of its privileges or to extract data from its memory space. Before executable code is commonly made read-only, an attacker could arbitrarily change the code as it is run, targeting direct transfers or even do with no transfers at all. After W^X became widespread, an attacker wants to instead redirect execution to a separate, unprotected area containing the code to be run, making use of indirect transfers: one could overwrite the virtual table for a forward-edge attack or change the call stack for a backward-edge attack (return-oriented programming). CFI is designed to protect indirect transfers from going to unintended locations.

Let's look at the below control flow diagram to understand it better:



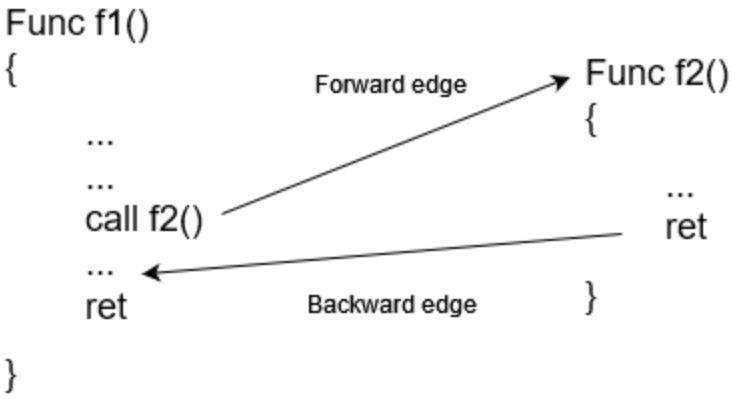
For functions from A to D, the default and legal control flow / call graph looks something like above. With CFG in place, any illegal control transfer will be blocked or detected during runtime.



The CFI can be implemented for one of the following cases:

- Forward edge Integrity
- Backward edge integrity

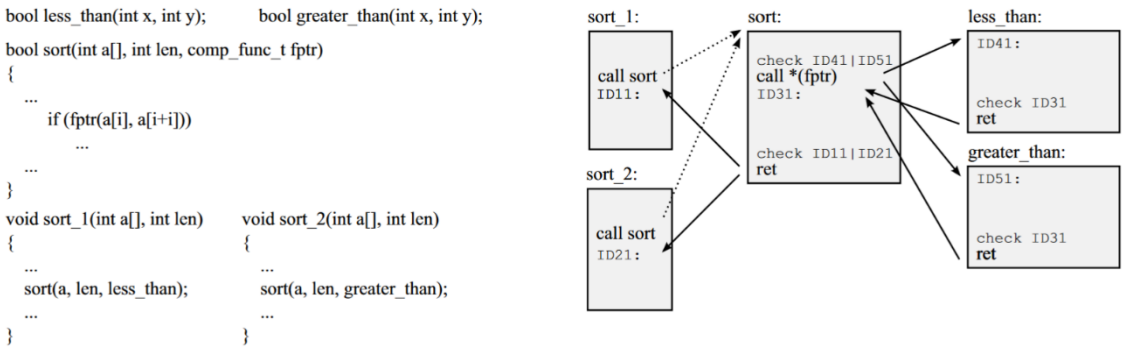
Both can be explained using below code illustration:



For the above high level implementation, Forward edge integrity will take care that function f1 will only call f2 or f2 is only called from f1. Whereas Backward edge integrity will take care that ret or function exit from f2 will only return to f1. For a successful CFI, both forward and backward edges would need to be protected to maintain control flow integrity and prevent attackers from diverting the program's execution in any direction.

Initial CFI implementations

In early stage CFI implementations like CCFIR and bin-CFI, Every instruction that is the target of a legitimate control-flow transfer is assigned a unique identifier (ID), and checks are inserted before control-flow instructions to ensure that only valid targets are allowed. Direct transfers have a fixed target and they do not require any enforcement checks. However, indirect transfers, like function calls and returns, and indirect jumps, take a dynamic target address as argument. As the target address could be controlled by an attacker due to a vulnerability, CFI checks to ensure that its ID matches the list of known and allowable target IDs of the instruction. The implementation can be understood better using below control flow graph:



CFI introduces labels and checks for all indirect transfers. Control-flow transfers checked by CFI are shown in solid lines.

Source: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6956588

In the year 2005, the initial surface of CFI research became publicly accessible. Subsequently, numerous additional researchers emerged over the course of several years, presenting evidence of various forms of control flow integrity. However, the primary release of CFI that was readily accessible occurred in 2014 for both Windows and Linux operating systems, with both the compilers of Windows and Linux incorporating some level of support for CFI. In 2014, Linux implemented its first CFI implementation, drawing upon the research conducted by the Google team as outlined in their paper titled "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM". Following suit, Windows also introduced CFI support in November 2014, referring to it as Control Flow Guard. Given that the Linux implementation gained public exposure first, let us delve into the CFI implementation within the Linux system.

> Note: We will first go through forward edge CFI and then backward edge CFI

Forward edge CFI

Forward edge CFI in linux

In 2014, few google researchers have implemented the first practical CFI implementation. This particular implementation was employed internally in conjunction with Chromium and a select few other products, prior to its public release in August of that year. Consequently, this CFI implementation was integrated into both the GCC and LLVM compilers during the same aforementioned year. This initial work contain two different methods for CFI:

- VTV – Virtual table verification
- IFCC – Indirect function-call checks

Both methods have been implemented in order to enforce the integrity of forward edges and primarily focus on safeguarding against arbitrary code execution resulting from memory corruption in the heap. The authors have observed that, although the process stack has already been adequately protected through the implementation of multiple mitigation measures up until 2014, the heap area of the process remains susceptible to memory corruptions, which can lead to arbitrary code execution without any significant mitigations in place. This has served as a motivation for the authors to incorporate compiler-based mechanisms, with the aim of further enhancing the protection and integrity of program control data, with particular emphasis on the integrity of control-transfer data stored in the heap.

VTV - Virtual table verification

The goal of VTV is to protect virtual calls in C++. The motivation behind this mitigation is to protect common types of code hijacking that happen in C++ based memory implementation due to heap exploitations.

From the name, you can predict that VTV is use to provide integrity of vtables present in C++ code. Since most indirect calls in C++ are through vtable, this mitigation is well suited for programs written in C++. We are not going to look into details of Vtable modifications works, but let’s summarize it at high level:

> The vtables themselves are placed in read-only memory, so they cannot be easily attacked. However, the objects making the calls are allocated on the heap. An attacker can make use of existing errors in the program, such as use-after-free, to overwrite the vtable pointer in the object and make it point to a vtable created by the attacker. The next time a virtual call is made through the object, it uses the attacker's vtable and executes the attacker's code.

To protect the above scenario, VTV verifies the validity, at each call site, of the vtable pointer being used for the virtual call, before allowing the call to execute. In particular, it verifies that the vtable pointer about to be used is correct for the call site, i.e., that it points either to the vtable for the static type of the object, or to a vtable for one of its descendant classes. The compiler passes to the verifier function the vtable pointer from the object and the set of valid vtable pointers for the call site. If the pointer from the object is in the valid set, then it gets returned and used. Otherwise, the verification function calls a failure function, which normally reports an error and aborts execution immediately.

VTV has two pieces: the main compiler part, and a runtime library (libvtv), both of which are part of GCC. In addition to inserting verification calls at each call site, the compiler collects class hierarchy and vtable information during compilation, and uses it to generate function calls into libvtv, which will (at runtime) build the complete sets of valid vtable pointers for each polymorphic class in the program.

To keep track of static types of objects and to find sets of vtable pointers, VTV creates a special set of variables called vtable-map variables (read only), one for each polymorphic class. At runtime, a vtable-map variable will point to the set of valid vtable pointers for its associated class. When VTV inserts a verification call, it passes in the appropriate vtable-map variable for the static type of the object, which points to the set to use for verification.

VTV in action

Consider the following program:

```
#include <stdio.h>
#include <iostream>
using namespace std;

class Animal // base class
{

public:
int weight;
virtual int getWeight() { return 12;};
virtual int getMass() { return 120;};
};


// Obviously, Tiger derives from the Animal class
class Tiger: public Animal {
```

```
public:

int weight;

int height;

int getWeight() {return weight;};

int getMass() { return height;};

int getHeight() {return height;};
};

int main()
{
        Tiger t1;

        /* below, an Animal object pointer is set to point
           to an object of the derived Tiger class  */

        Animal *a1 = &t1;

        /*  below, how does this know to call the
                definition of getWeight in the Tiger class,
                and not the definition provided in the Animal
                class  */

        cout << (a1->getMass());
}
```

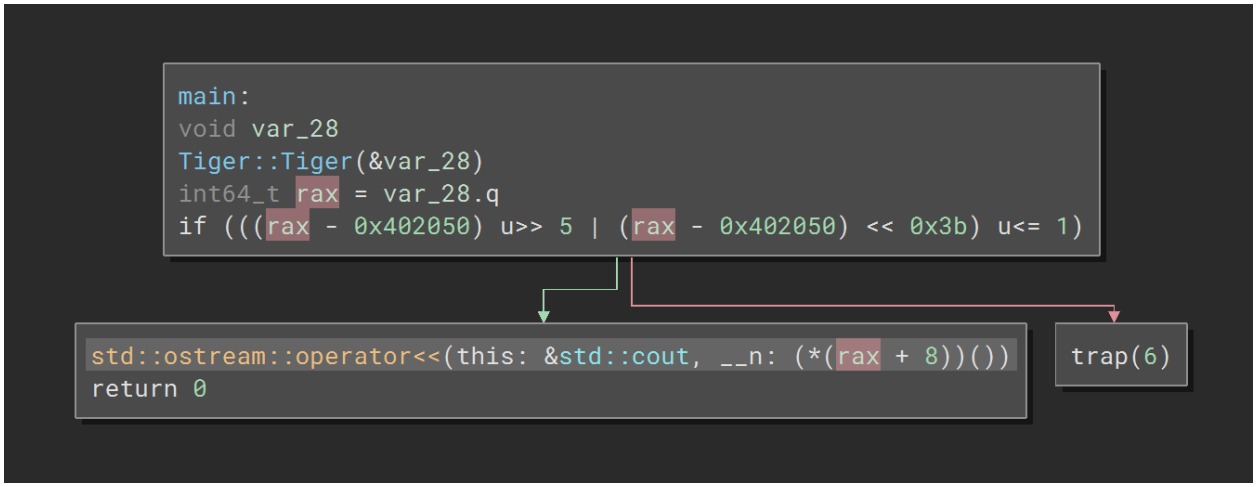
To use compile binary with VTV enable through clang use the following command:

```
clang++ -fsanitize=cfi-vcall -fvisibility=hidden -flto test2.cpp -o
test2_cfi
```

After compiling let's compare the code difference with and without the CFI-VTV enabled.

Before:

After:



In the second image you will notice that before calling *a1->getmass() (address [rax+8])*, it is been verified if it is in the range of valid call site which is added during IR phase of compilation. The check can be explained in more details here: https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html

IFCC: Indirect Function-Call Checks

Unlike VTV, IFCC mechanism protect integrity of all kinds of indirect calls. , it protects indirect calls by generating jump tables for indirect-call targets and adding code at indirect-call sites to transform function pointers, ensuring that they point to a jump-table entry. Any function pointer that does not point into the appropriate table is considered a CFI violation and will be forced into the right table by IFCC. The valid jump table entries are again created based on same function prototypes.

IFCC In action:

Let's look at the following C program:

```c
#include <stdio.h>

// Function prototypes
int add(int a, int b);

int main() {
    // Declare a function pointer that points to a function taking two int arguments and returning int
    int (*operation)(int, int);

    // Let's perform addition
    operation = add;
    int result = operation(10, 5);
    printf("Result of addition: %d\n", result);

    return 0;
}

// Function definitions
int add(int a, int b) {
    return a + b;
}
```

To compile the program with IFCC, use the following command:

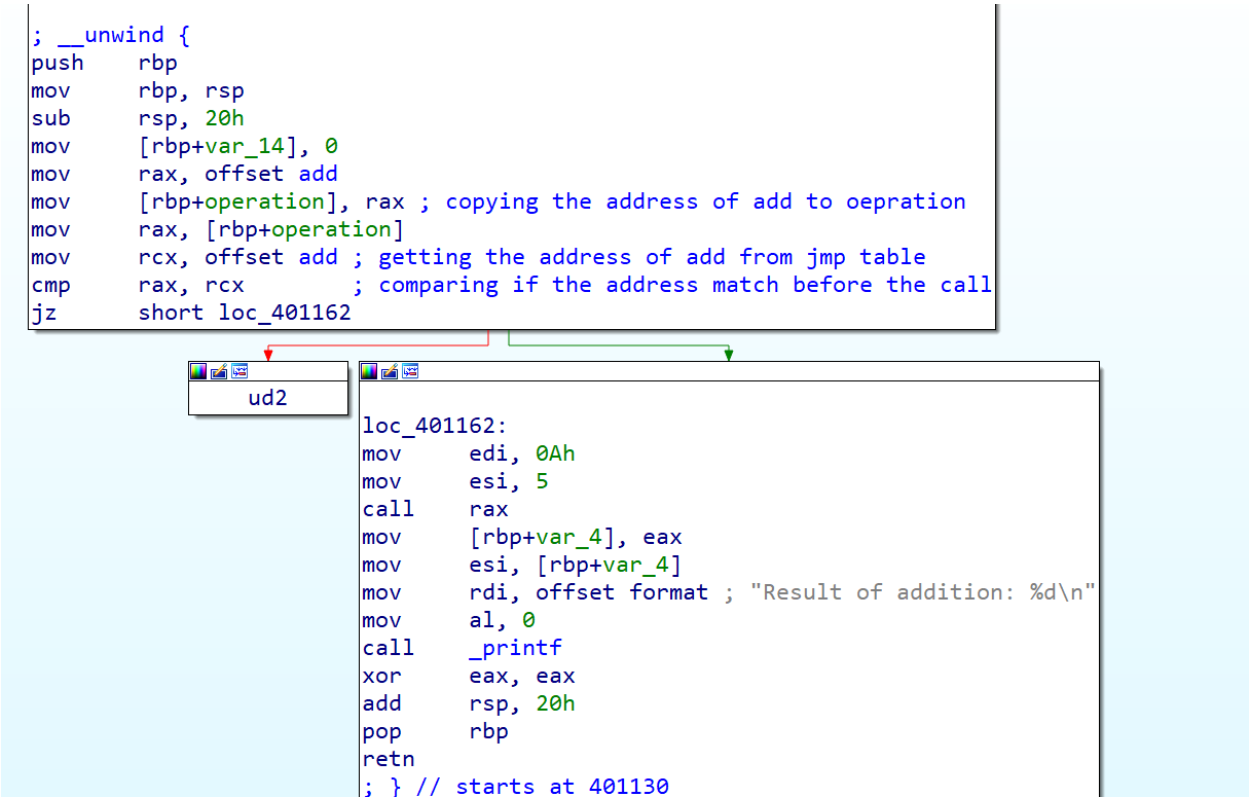
```
clang -fsanitize=cfi-icall -flto indirect_call.c -o indirect_call_cfi
```

Let's compare the assembly sequence of both cases in IDA:

Before

```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_18= dword ptr -18h
result= dword ptr -14h
operation= qword ptr -10h
var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+var_4], 0
mov     rax, offset add
mov     [rbp+operation], rax ; coping the address of add
mov     edi, 0Ah
mov     esi, 5
call    [rbp+operation] ; call add
mov     [rbp+result], eax
mov     esi, [rbp+result]
mov     rdi, offset format ; "Result of addition: %d\n"
mov     al, 0
call    _printf
xor     ecx, ecx
mov     [rbp+var_18], eax
mov     eax, ecx
add     rsp, 20h
pop     rbp
retn
; } // starts at 401130
```

After:



In the second case where IFCC is enabled, before calling add through operation (*call rax*), the address is first verified if matches the address in the jmp table (*mov rcx, offset add, cmp rax, rcx*) or not. Since we only have 1 address in the jump table, the check is straight forward but in case of multiple entries, you will see *ror* (rotate) instruction to verify if the jump index is within range.

Other CFI support in clang:

Clang also support few other schemes that enhance the overall CFI mitigation. These schemes mostly rely on verifying function prototype before jumping to that location. Listed in detail below.

`-fsanitize=cfi-cast-strict` – If a class has a single non-virtual base and does not introduce or override virtual member functions or fields other than an implicitly defined virtual destructor, it will have the same layout and virtual function semantics as its base. By default, casts to such classes are checked as if they were made to the least derived such class.

`-fsanitize=cfi-derived-cast` and `-fsanitize=cfi-unrelated-cast` – These scheme checks that pointer casts are made to an object of the correct dynamic type; that is, the dynamic type of the object must be a derived class of the pointee type of the cast. The checks are currently only introduced where the class being casted to is a polymorphic class. First one is bad casts from a base class to a derived class and 2nd one is bad casts from a pointer of type void* or another unrelated type.

`-fsanitize=cfi-nvcall` – This scheme checks that non-virtual calls take place using an object of the correct dynamic type; that is, the dynamic type of the called object must be a derived class of the static type of the object used to make the call.

`-fsanitize=cfi-mfcall` - This scheme checks that indirect calls via a member function pointer take place using an object of the correct dynamic type. Specifically, we check that the dynamic type of the member function referenced by the member function pointer matches the "function pointer" part of the member function pointer, and that the member function's class type is related to the base type of the member function.

To compile the code with all the CFI mitigation in place, you can use following flag: `-fsanitize=cfi`

Resources:

https://clang.llvm.org/docs/ControlFlowIntegrity.html

CFI in linux kernel

Forward edge CFI was first introduced in android linux kernel upstream in 2018 which was later added in linux kernel upstream in 2021. The first implementation has support for IFCC from CLANG in linux kernel. It can be controlled using *CONFIG_CFI_CLANG* and once set the compiler injects a runtime check before each indirect function call to ensure the target is a valid function with the correct static type. With CFI enabled, the compiler injects a *__cfi_check()* function into the kernel and each module for validating local call targets. Similar to what we seen in IFCC, during linux kernel compilation clang implements indirect call checking using jump tables and offers two methods of generating them. With canonical jump tables, the compiler renames each address-taken function to *<function>.cfi* and points the original symbol to a jump table entry, which passes *__cfi_check()* validation.

You can check the PR here:
https://github.com/torvalds/linux/commit/cf68fffb66d60d96209446bfc4a15291dc5a5d41

kCFI – The fine grained CFI scheme for linux kernel

In 2022 kernel version 6.1.X, linux have merged a new patch for CFI called kCFI which is a fine grained CFI scheme that overcomes almost all the major issues and limitations of earlier CFI implementation from CLANG. kCFI's main goal is to improve fine grained CFI scheme for indirect call redirection issues in linux kernel. It is totally dependent on instrumentation, and requires no runtime component. This decreases the overhead that is usually seen with most CFI based implementations. kCFI provides both forward edges as well as backward edge but we will only look at examples of forward edge CFI below. But the backward edge (return guard) is implemented in a similar way. kCFI over-approximates the call graph by considering valid targets for an indirect call all those functions that have a matching prototype with the pointer used in the indirect call.

Let's go through the original LLVM-CFI issues one by one that kCFI overcome for linux kernel:

- Limitation 1: **Performance bottleneck due to jump table based CFI implementation**

Jump table based IFCC implementation comes with significant performance overhead when running in linux kernel. kCFI overcomes this by having tag based assertions. Tag based CFI check can be understood with following example:

(a) Prologue(s) instrumentation (tag).

```
1  ...
2  <func>:
3  nopl    0xbcbee9
4  ...
```

(b) Indirect call site(s) instrumentation (guard).

```
1  ...
2  cmpl    $0xbcbee9,0x4(%rax)
3  je      <7>
4  push    %rax
5  callq   <kcfi_vhndl>
6  pop     %rax
7  callq   *%rax
8  nopl    0x138395f
9  ...
```

Unlike LLVM CFI, kCFI adds tags using long nop instructions and verify them before the call to an indirect function. In above snippet, prologue func has an entry-point tag that is verified at call site (b) using *cmpl* instruction(since *rax* will have the address of func). this snippet dereferences *0x4(%rax)* and compares the result with the expected ID (*0xbcbee9*; line 2). If the two IDs match, the control jumps to the *callq* instruction and the indirect invocation of func takes place (lines 3 and 7); else, the bogus branch address is pushed onto the stack and *kcfi_vhndl* (violation handler) is invoked (lines 4–6).

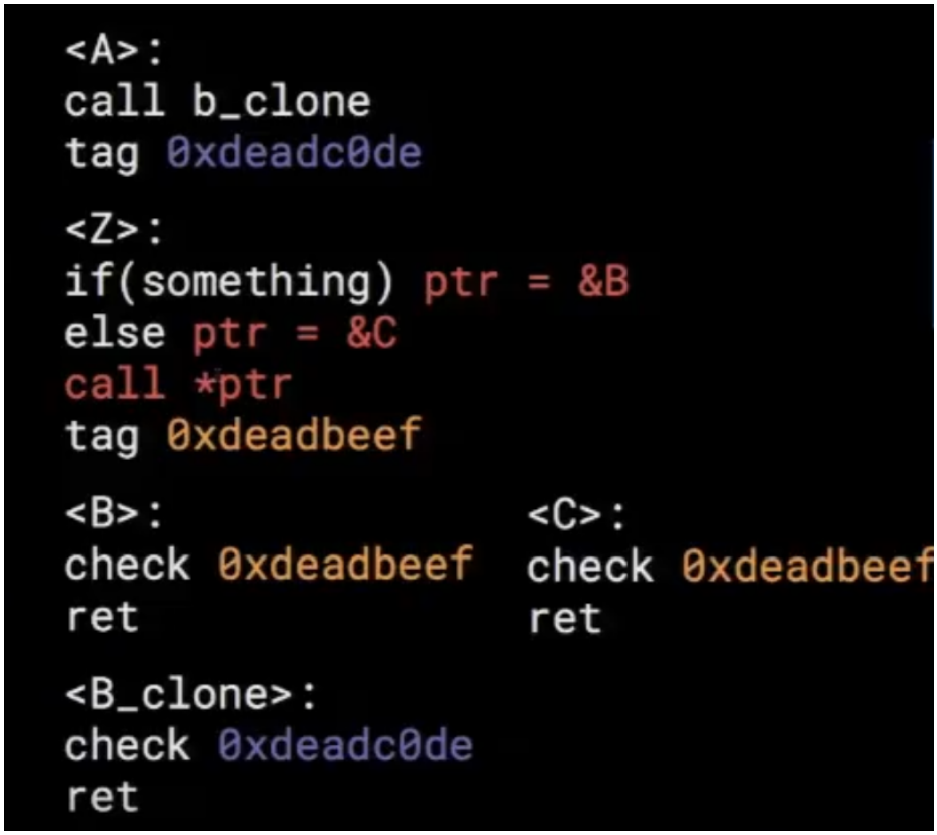
- Limitation 2: **There exist a vast multitude of kernel functions that possess a similar prototype, such as *void foo(void)*, which renders them eligible as CFI targets for one another.**

To reduce similar valid call site jumps, kCFI introduced call graph detaching. It can be understood using below example:

```
<A>:
call b
tag 0xdeadbeef

<Z>:
if(something) ptr = &B
else ptr = &C
call *ptr
tag 0xdeadbeef

<B>:                <C>:
check 0xdeadbeef    check 0xdeadbeef
ret                 ret
```

In the above code function B and C have same tag due to similar prototype, so does A due to direct call to B. In such case C is allowed to return to A even through it's not legal. This creates a situation where transitively all instructions after a direct call to a function become valid return points to other functions with a similar prototype. This makes CFI prone to something called a bending attack.

To mitigate this problem, kCFI follows a novel approach by cloning functions instead of merging all valid return targets. In this way, a function named *foo()* is cloned into a new function called *foo_direct()*, which has the same semantics but checks for a different tag before returning. All direct calls to *foo()* are then replaced by calls to *foo_direct()*, and the tag placed after the call site is the one that corresponds to *foo_direct()*. This can be understood more easily with below illustration:



In the updated code with CGD in place, you will see A calling *B_clone* (having different tag) rather than B. Now, C will not be able to return to A due to tag mismatch.

- Limitation 3: **Support for self-modifying code and LKMs**

By employing tag-based assertions, kCFI supports self-modifying code and LKMs, as long as these portions of code are compiled in a compatible way.

- Limitation 4: **Support for inline assemble code**

One of the drawbacks of using LLVM-based instrumentation is that assembly sources are not touched, as this kind of code is directly translated into binaries without having an intermediate representation (IR) form. The kernel has a significant part of its code written in assembly, which includes many indirect branches. While applying CFI, if such code is left unprocessed, two major problems arise: (i) indirect

branches in assembly sources are left unprotected, and (ii) tags are not placed, breaking compatibility with C functions returning to assembly, or with assembly functions being called indirectly from C code. kCFI tackles this problem through the automatic rewriting of the assembly sources assisted by information extracted during code and binary analysis.

Resources:

You can check the changelog of patch here:
https://github.com/torvalds/linux/commit/865dad2022c52ac6c5c9a87c5cec78a69f633fb6

You can read about kCFI here:
https://www.blackhat.com/docs/asia-17/materials/asia-17-Moreira-Drop-The-Rop-Fine-Grained-Control-Flow-Integrity-For-The-Linux-Kernel-wp.pdf

CFG (Control flow guard):

Unlike other mitigation techniques that gained popularity for their implementation in Linux, CFI gained popularity for its implementation in Windows in 2014 (in Windows 8.1). It was later removed but was reintroduced with changes in the Windows 10 Anniversary update 14393. While Windows's CFG initially received significant attention upon its release, it is limited in terms of capabilities and coverage compared to CLANG's CFI implementation. Additionally, it has faced considerable criticism due to the constant discovery of bypasses by security researchers.

Sole focus of CFG is to protect the integrity of indirect function calls in a somewhat similar way as IFCC. Let's look at the internal details of how the CFG is implemented in windows.

CFG Internals

If the */cfguard* flag is used with the msvc compiler (Visual Studio compiler), it will enable CFG when compiling the binary. The resulting binary will include a data directory called *Load Configuration* directory, which contains the CFG configuration details for the binary. Load Configuration directory have a structure which have few fields that are important for CFG implementation:

```
    ULONGLONG  GuardCFCheckFunctionPointer;     // VA
    ULONGLONG  GuardCFDispatchFunctionPointer; // VA
    ULONGLONG  GuardCFFunctionTable;           // VA
    ULONGLONG  GuardCFFunctionCount;
    DWORD      GuardFlags;
```

GuardFlags have the flags related to CFG which define what CFG mitigations are set in binary. The structure looks like below:

```
#define IMAGE_GUARD_CF_INSTRUMENTED 0x00000100 // Module performs
control flow integrity checks using system-supplied support
#define IMAGE_GUARD_CFW_INSTRUMENTED 0x00000200 // Module performs
control flow and write integrity checks
#define IMAGE_GUARD_CF_FUNCTION_TABLE_PRESENT 0x00000400 // Module
contains valid control flow target metadata
#define IMAGE_GUARD_SECURITY_COOKIE_UNUSED 0x00000800 // Module does
not make use of the /GS security cookie
#define IMAGE_GUARD_PROTECT_DELAYLOAD_IAT 0x00001000 // Module
supports read only delay load IAT
#define IMAGE_GUARD_DELAYLOAD_IAT_IN_ITS_OWN_SECTION 0x00002000 //
Delayload import table in its own .didat section (with nothing else
in it) that can be freely reprotected
#define IMAGE_GUARD_CF_EXPORT_SUPPRESSION_INFO_PRESENT 0x00004000 //
Module contains suppressed export information. This also infers that
the address taken

// taken IAT table is also present in the load config.
#define IMAGE_GUARD_CF_ENABLE_EXPORT_SUPPRESSION 0x00008000 // Module
enables suppression of exports
#define IMAGE_GUARD_CF_LONGJUMP_TABLE_PRESENT 0x00010000 // Module
contains longjmp target information
#define IMAGE_GUARD_RF_INSTRUMENTED 0x00020000 // Module contains
return flow instrumentation and metadata
#define IMAGE_GUARD_RF_ENABLE 0x00040000 // Module requests that the
OS enable return flow protection
#define IMAGE_GUARD_RF_STRICT 0x00080000 // Module requests that the
OS enable return flow protection in strict mode
#define IMAGE_GUARD_RETPOLINE_PRESENT 0x00100000 // Module was built
with retpoline support
#define IMAGE_GUARD_CF_FUNCTION_TABLE_SIZE_MASK 0xF0000000 // Stride
of Guard CF function table encoded in these bits (additional count of
bytes per element)
#define IMAGE_GUARD_CF_FUNCTION_TABLE_SIZE_SHIFT 28 // Shift to
right-justify Guard CF function table stride
```

Functions that are valid indirect call targets are listed in the *GuardCFFunctionTable*, sometimes termed the GFIDS table. This is a sorted list of relative virtual addresses (RVA) that contain information about valid CFG call targets. There are two other function pointer with following usecase:

GuardCFCheckFunctionPointer provides the address of an OS-loader provided symbol that can be called with a function pointer in the first integer argument register (ECX on x86) which will return on success or will abort the process if the call target is not a valid CFG target.

The *GuardCFDispatchFunctionPointer* provides the address of an OS-loader provided symbol that takes a call target in register RAX and performs a combined CFG check and tail branch optimized call to the call target (registers R10/R11 are reserved for use by the *GuardCFDispatchFunctionPointer* and integer argument registers are reserved for use by the ultimate call target).

In addition to possessing a specific header for CFG, Windows executes two additional tasks in order to enable CFG for a binary:

- Instrument around all indirect call with *_guard_check_icall* check.
- Mapping CFG bitmap in process memory space during Process initialization

***__guard_dispatch_icall_fptr* check**

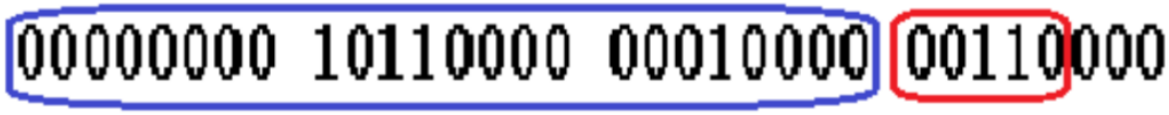
CFG when enabled, msvc compiler will wrap all the indirect calls in a given binary by a call to *__guard_dispatch_icall_fptr* (Guard CF address dispatch-function pointer) which ensure the target address is valid. The wrapper function *_guard_dispatch_icall_fptr* is actually a placeholder at compile-time and will be patched by the NT loader on module loading to point to *LdrpValidateUserCallTarget* to do the actual check. We will look into the call in the next section for better understanding.

CFG Bitmap

The NT loader will, on a module load (see `ntdll!LdrSystemDllInitBlock`), parse the `Load Configuration` entry to look for `CFG` aware capabilities and, if enabled, will generate a `CFG` bitmap storing all the valid targets address from the `CFG` whitelist in the module. *__guard_dispatch_icall_fptr* calls *ntdll!LdrpValidateUserCallTarget* which during execution verifies the call to be valid using CFG Bitmap loaded in memory.

CFGBitmap represents the starting location of all the functions in the process space. The status of every 8 bytes in the process space corresponds to a bit in CFGBitmap. If there is a function starting address in each group of 8 bytes, the corresponding bit in CFGBitmap is set to 1; otherwise it is set to 0.

Let's take the function target address to be 0x00b01030. The address is used to get the bit in Bitmap and verified if is 1 or 0.



The highest 3 bytes (the 24 bits encircled in blue) is the offset for CFGBitmap (unit is 4 bytes/32 bits). In this example, the highest three bytes are equal to 0xb010.Therefore, the pointer to a four byte unit in CFGBitmap is the base address ofCFGBitmap plus 0xb010.

Meanwhile, the fourth bit to the eighth bit (the five bits encircled in red) have the value X. If the target address is aligned with 0x10 (target address & 0xf == 0), then X is the bit offset value within the unit. If the target address is not aligned with 0x10(target address & 0xf != 0), the X | 0x1 is the bit offset value. If the bit is equal to 1, it means the indirect call target is valid because it is a function's starting address. If the bit is 0, it means the indirect call target is invalid because it is not a function's starting address.

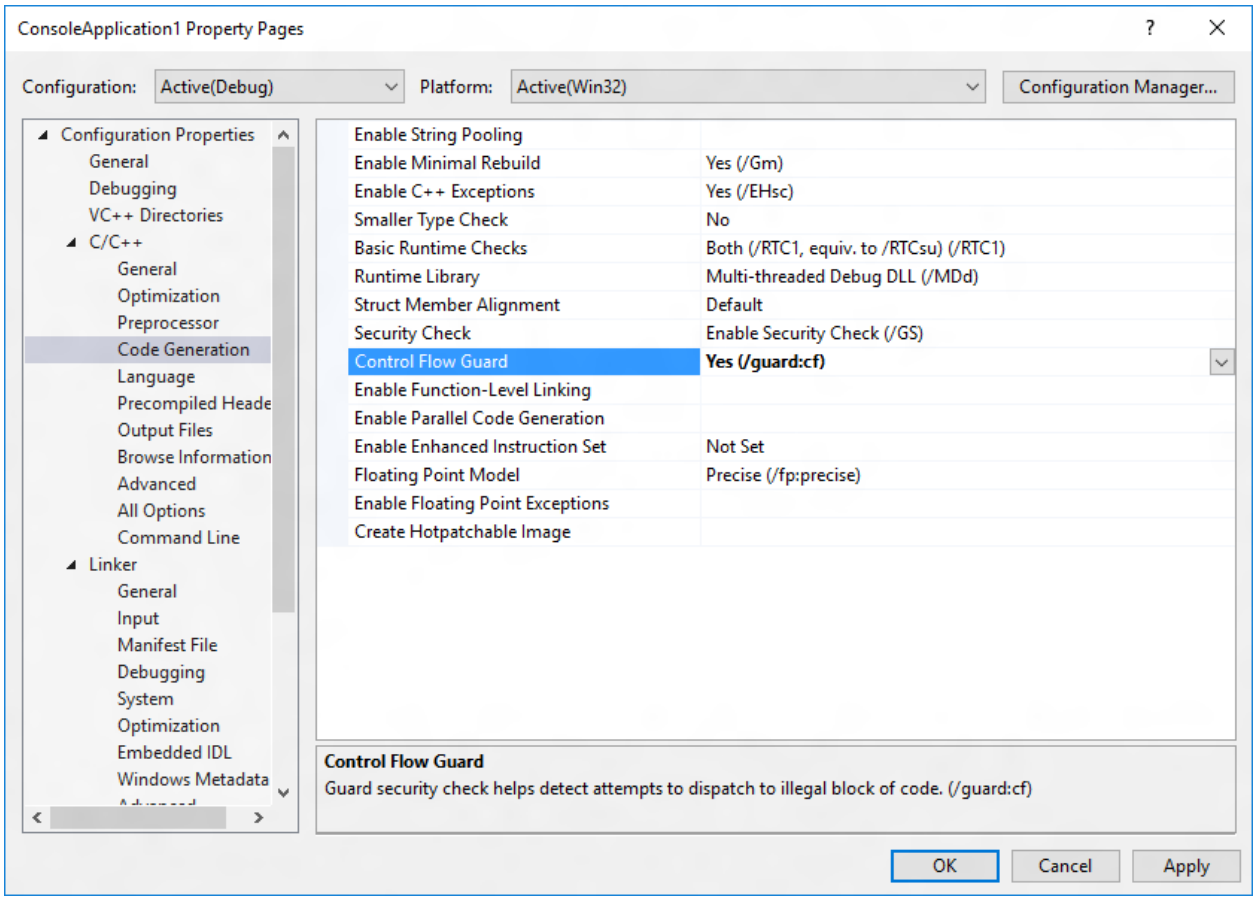
CFG in Action

Let's compile the following program with CFG enable to check the modifications in binary due to CFG.

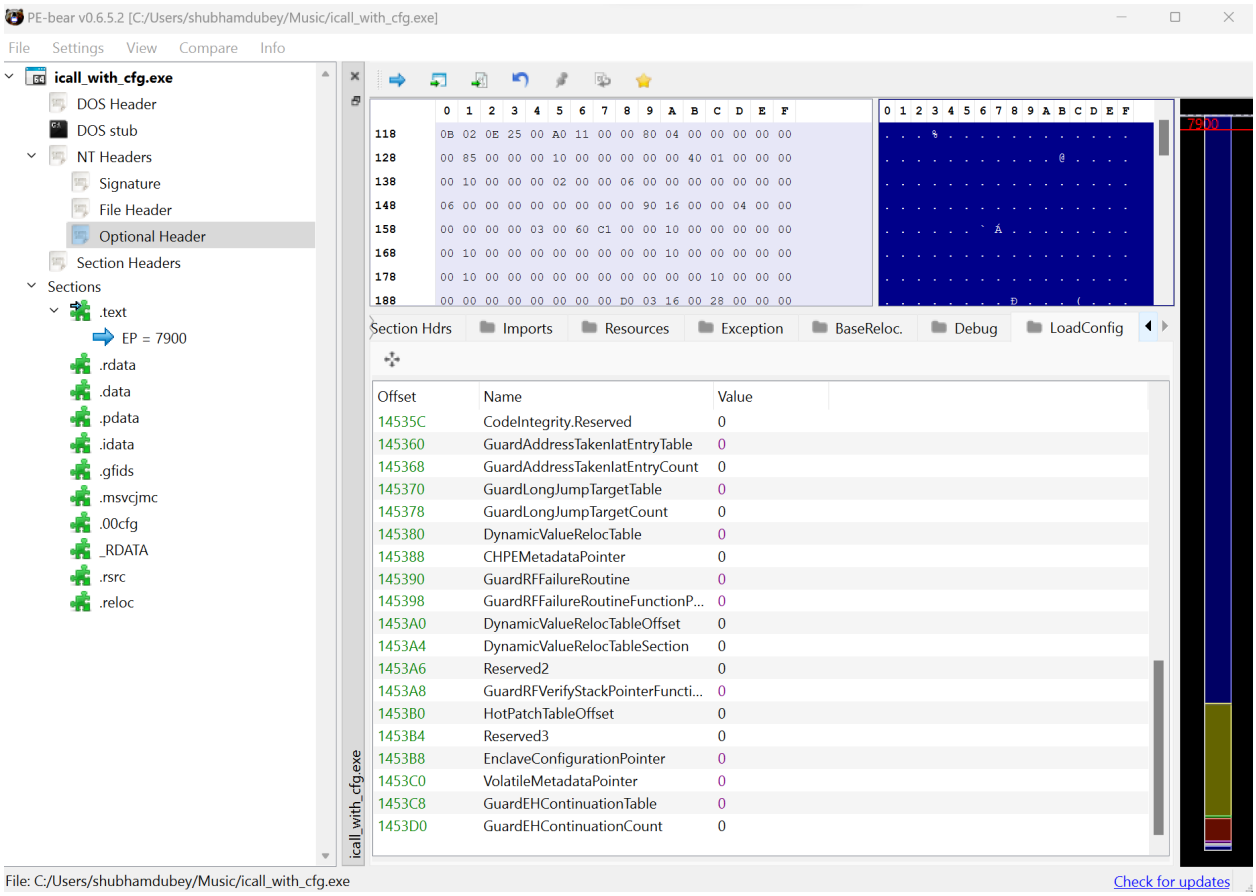
```
#include <stdio.h>
#include<Windows.h>
#include<Memoryapi.h>
// Function prototypes
int add(int a, int b);
int subtract(int a, int b);

int main() {
    // Declare a function pointer
    int a;

    int (*operation)(int, int);
    printf("Enter 1 for add and 2 for substract:");
    scanf_s(" %d", &a);
    if (a == 1)
    {
        // Assign the address of the add function to the function pointer
        operation = add;

        int result1 = operation(5, 3);
        printf("\nResult of addition: %d\n", result1);
    }
    else
    {
        // Assign the address of the subtract function to the function pointer
        operation = subtract;

        // Call the function indirectly using the function pointer
        int result2 = operation(5, 3);
        printf("\nResult of subtraction: %d\n", result2);
    }


    return 0;
}
```

```c
// Define the add function
int add(int a, int b) {
    return a + b;
}

// Define the subtract function
int subtract(int a, int b) {
    return a - b;
}
```

You can turn on CFG using following visual studio configuration options

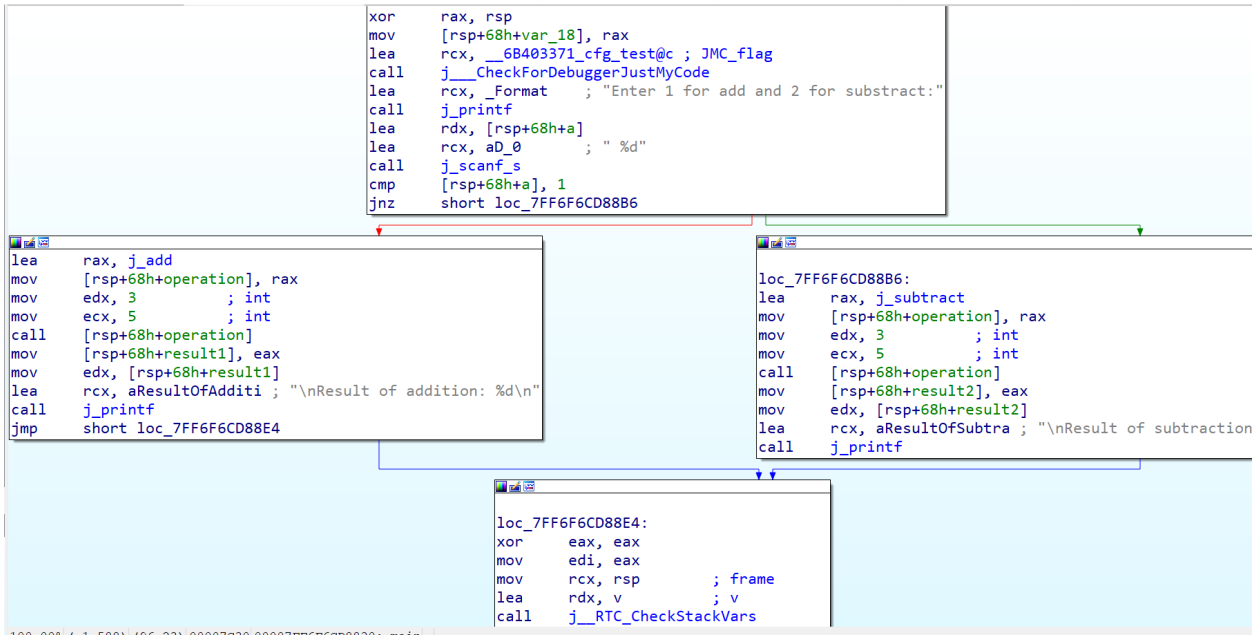


Once the binary is compiled, you can load it in PEBear to verify it has a Load Configuration data directory.

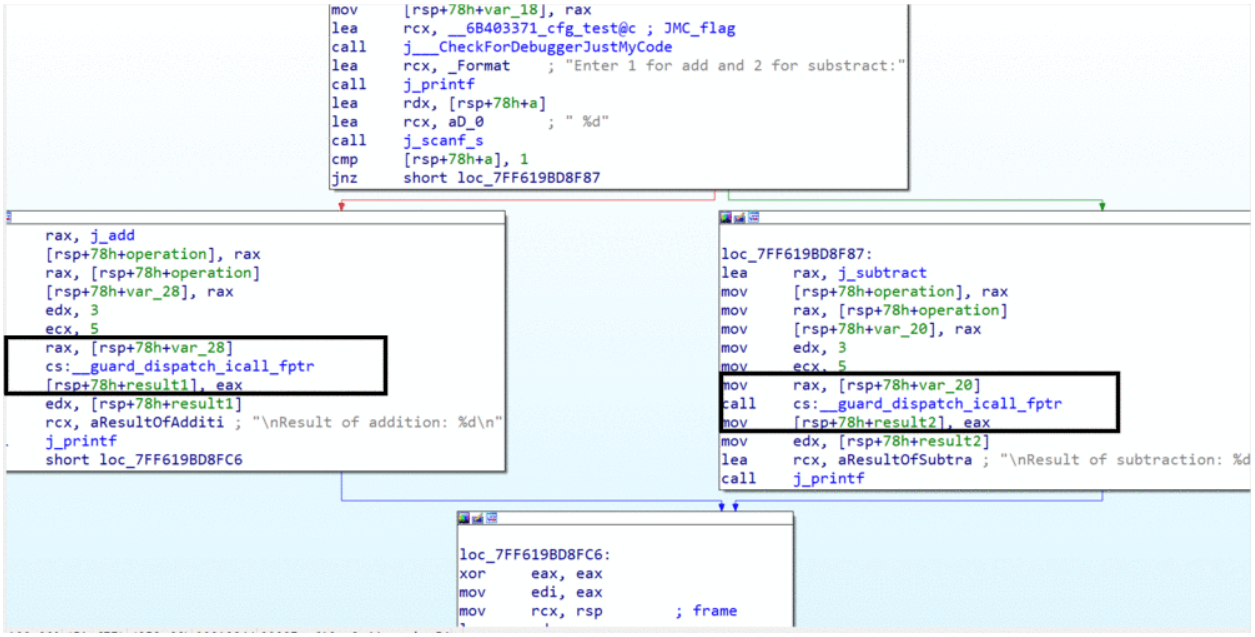


Now, let's load the binary in IDA and look at the changes.

Before



After:



You can notice that the indirect call (*call rsp+68h+operation*) gets replaced with call to *_guard_dispatch_icall_fptr*. The calling address is passed using *rax* register and variables are passed using the default calling convention *rcx, rdx*…

The code for verifying the call target using bitmap is present in ntdll:

```
ntdll.dll:00007FFD92C7EBC0                mov     r11, cs:off_7FFD92D893C8 ; base address of bitmap
ntdll.dll:00007FFD92C7EBC7                mov     r10, rax        ; address of add function
ntdll.dll:00007FFD92C7EBCA                shr     r10, 9
ntdll.dll:00007FFD92C7EBCE                mov     r11, [r11+r10*8]
ntdll.dll:00007FFD92C7EBD2                mov     r10, rax
ntdll.dll:00007FFD92C7EBD5                shr     r10, 3
ntdll.dll:00007FFD92C7EBD9                test    al, 0Fh         ; test least significant 4 bits
ntdll.dll:00007FFD92C7EBDB                jnz     short loc_7FFD92C7EBE6
ntdll.dll:00007FFD92C7EBDD                bt      r11, r10        ; check if specific bit in bitmap is set
ntdll.dll:00007FFD92C7EBE1                jnb     short loc_7FFD92C7EBF1
ntdll.dll:00007FFD92C7EBE3                jmp     rax
```

If the check fails, the execution get stopped.

Resources

http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf

https://lucasg.github.io/2017/02/05/Control-Flow-Guard/

https://learn.microsoft.com/en-us/windows/win32/secbp/pe-metadata

XFG (extended flow guard) (Honorable mention)

Due to coarse gain nature of CFG where attacker can still call the gadgets part of valid call sites, Windows developer's decided to come up with a fine grained solution on top of CFG

XFG adds a check on top of default CFG which verifies if the caller who called the call site is correct or not. To perform this, the compiler will generate a 55-bit hash based on the function name, number of arguments, the type of arguments, and the return type. This hash will be embedded in the code just prior to the call into XFG. Later inside *LdrpDispatchUserCallTargetXFG* the value is matched to be the same or not.

At the caller you will see something like below:

```
main+20       call    printf
main+25       mov     rax, [rsp+38h+var_18]
main+2A       mov     [rsp+38h+var_10], rax
main+2F       mov     r10, 99743F3270D52870h
main+39       movss   xmm1, cs:__real@40000054
main+41       movss   xmm0, cs:__real@3f800054
main+49       mov     rax, [rsp+38h+var_10]
main+4E       call    cs:__guard_xfg_dispatch_icall_fptr
main+54       xor     eax, eax
main+56       add     rsp, 38h
main+5A       retn
```

mov at main+2f moves the generated hash to r10. Inside *_guard_xfg_dispatch_icall_fptr* you will see the following:

```
LdrpDispatchUserCallTargetXFG      LdrpDispatchUserCallTargetXFG proc near
LdrpDispatchUserCallTargetXFG      ; __unwind { // LdrpICallHandler
LdrpDispatchUserCallTargetXFG          or      r10, 1
LdrpDispatchUserCallTargetXFG+4        test    al, 0Fh
LdrpDispatchUserCallTargetXFG+6        jnz     short loc_180094337
LdrpDispatchUserCallTargetXFG+8        test    ax, 0FFFh
LdrpDispatchUserCallTargetXFG+C        jz      short loc_180094337
LdrpDispatchUserCallTargetXFG+E        cmp     r10, [rax-8]
LdrpDispatchUserCallTargetXFG+12       jnz     short loc_180094337
LdrpDispatchUserCallTargetXFG+14       jmp     rax
```

The hash value (present in *r10*) is compared against the original generated value [*rax-8*] at the end of the function before calling the actual target.

Hardware enforced CFI mitigations

CFI mitigation gained rapid popularity following its initial integration into major platforms. However, both mitigations are disabled by default in all compilers due to the increased performance overhead. At this juncture, hardware vendors have taken it upon themselves to ensure that CFI is readily accessible. Intel and ARM both has introduced similar kind of mitigation for forward edge integrity, explained below.

BTI (branch target identification)

In the year 2018, ARM unveiled the initial hardware-enforced forward edge CFI within the ARM 8.5-A processor lineage, which was named BTI (Branch target identification). The primary objective of BTI is to forestall indirect calls from redirecting to unintended destinations, thus hindering the execution of gadgets.

BTI technical details

BTI is straightforward in terms of its implementation. If BTI is enabled, the first instruction encountered after an indirect jump must be a special BTI instruction. When BTI is turn off, this first instruction will be treated as no-op. When BTI is on, the processor check if the BTI instruction is present as the first instruction or not. Jumps to locations that do not feature a BTI instruction, instead, will lead to the quick death of the process involved.

During branching, the type of branch is stored in the *PSTATE BTYPE* bits. Upon reaching the destination address, the processor checks whether the first instruction is BTI or not and verifies if the value passed as operand of BTI instruction matches with *PSTATE BTYPE* or not.

BTI in action

Let’s compile following program to test BTI compiled code.

```
#include<stdio.h>
#include<stdlib.h>

int add(int a, int b);
int subtract(int a, int b);

int main() {
    // Declare a function pointer
    int a;

    int (*operation)(int, int);
    printf("Enter 1 for add and 2 for substract:");
    scanf(" %d", &a);
    if (a == 1)
    {
        // Assign the address of the add function to the function pointer
```

```
        operation = add;

        int result1 = operation(5, 3);
        printf("\nResult of addition: %d\n", result1);
    }
    else
    {
        // Assign the address of the subtract function to the function
pointer
        operation = subtract;

        // Call the function indirectly using the function pointer
        int result2 = operation(5, 3);
        printf("\nResult of subtraction: %d\n", result2);
    }


    return 0;
}

// Define the add function
int add(int a, int b) {
    return a + b;
}

// Define the subtract function
int subtract(int a, int b) {
    return a - b;
}
```

Compile the program with the following parameters in ARM gcc.

```
gcc -mbranch-protection=bti ibt_arm.c -o ibt_arm
```

check the compiled code

```
000000000000086c <add>:
 86c:   d503245f        bti     c
 870:   d10043ff        sub     sp, sp, #0x10
 874:   b9000fe0        str     w0, [sp, #12]
 878:   b9000be1        str     w1, [sp, #8]
 87c:   b9400fe1        ldr     w1, [sp, #12]
 880:   b9400be0        ldr     w0, [sp, #8]
 884:   0b000020        add     w0, w1, w0
 888:   910043ff        add     sp, sp, #0x10
 88c:   d65f03c0        ret

0000000000000890 <subtract>:
 890:   d503245f        bti     c
 894:   d10043ff        sub     sp, sp, #0x10
 898:   b9000fe0        str     w0, [sp, #12]
 89c:   b9000be1        str     w1, [sp, #8]
 8a0:   b9400fe1        ldr     w1, [sp, #12]
 8a4:   b9400be0        ldr     w0, [sp, #8]
 8a8:   4b000020        sub     w0, w1, w0
 8ac:   910043ff        add     sp, sp, #0x10
 8b0:   d65f03c0        ret
 8b4:   d503201f        nop
 8b8:   d503201f        nop
```

You will notice the first instruction to be replaced as BTI in above assembly snippet. The syntax for BTI is

BTI <branch type>

There are 3 variants of the BTI instruction, which are valid targets for different kinds or branches: -

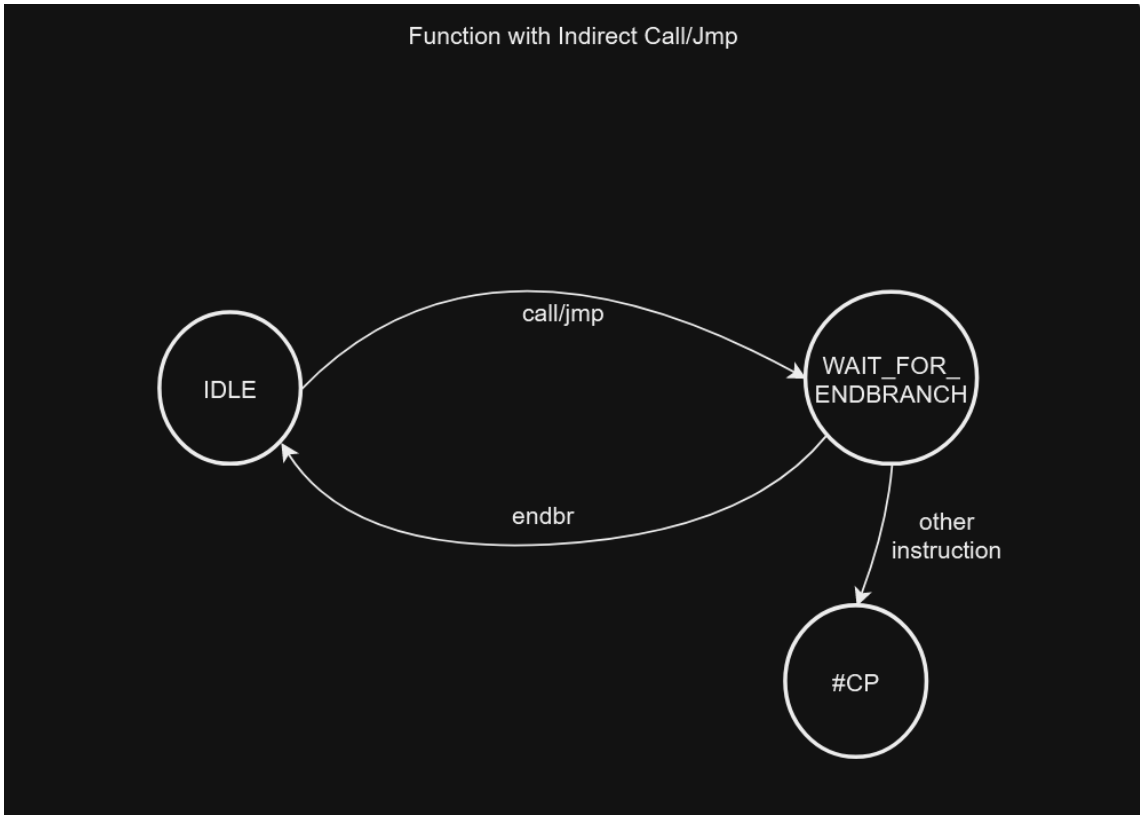
- ***c*** -Branch Target Identification for function calls
- ***j*** - Branch Target Identification for jumps
- ***jc*** - Branch Target Identification for function calls or jumps.

In our case the value is "*BTI c*" since *add* and *sub* are called as indirect function calls.

IBT (Indirect Branch Tracking)

In the year 2020, Intel unveiled the particulars of the introduction of a hardware security measure called CET for Intel TigerLake processors, which became accessible to the publically in 2021. Intel CET encompasses two hardware-enforced measures referred to as Shadow stack and IBT (Indirect branch tracking). IBT represents one of the methods employed to mitigate the issue of forward edge CFI.

Similar to BTI, if IBT is enabled, the CPU will ensure that every indirect branch lands on a special instruction (*endbr32* or *endbr64*), which executes as a no-op. If processor finds any other instruction than the expected *endbr*, it will raise a control-protection (#CP) exception. The state of IBT can be understood using following state machine:



The processor implements a state machine that tracks indirect *JMP* and *CALL* instructions. When one of these instructions is seen, the state machine moves from *IDLE* to *WAIT_FOR_ENDBRANCH* state. In *WAIT_FOR_ENDBRANCH* state the next instruction in the program stream must be an *ENDBRANCH*. If an *ENDBRANCH* is not seen the processor causes a control protection fault (#CP), otherwise the state machine moves back to IDLE state.

IBT in action

Let's compile the same program we used for BTI for IBT:

```
gcc -fcf-protection ibt_intel.c -o ibt_intel
```

You will notice following calls:

```
public subtract
subtract proc near

var_8= dword ptr -8
var_4= dword ptr -4

; __unwind {
endbr64
push    rbp
mov     rbp, rsp
mov     [rbp+var_4], edi
mov     [rbp+var_8], esi
mov     eax, [rbp+var_4]
sub     eax, [rbp+var_8]
pop     rbp
retn
; } // starts at 120B
subtract endp
```

Here, the first instruction is "*endbr64*" in all indirect calls.

In windows, you can compile binary with IBT by using the following flag in visual studio */CETCOMPAT*.

FineIBT (Honorable mention)

The default IBT implementation came with the drawback of allowing Code reuse of functions that are part of the Indirect branch target to be used as gadgets. To overcome this IBT limitation, In 2021 Intel's Joao Moreira raised patches for linux kernel which was later merged in linux kernel in 2022 https://github.com/torvalds/linux/commit/931ab63664f02b17d2213ef36b83e1e50190a0aa.

Under IBT, an attacker who is able to tamper with forward-edge transfers can still "bend" the control flow towards any of the valid/allowed function-entry points marked with *endbr*, because the CPU cannot differentiate among different types of endbr-marked code locations. To make a robust CFI solution using hardware assisted IBT, FineIBT instruments both the callers and the callees involved in indirect forward-edge transfers and verify if the correct callee is called from a given caller. The instrumentation can be understood using the following assembly snippet.

```
1  main:                        /* caller */
2    ...
3    mov   $0xc00010ff, %eax    /* SID = 0xc00010ff */
4    call  *%rcx
5    ...
6    call  func1_entry
7    ...
8  func0:                       /* callee */
9    endbr64
10   sub   $0xc00010ff, %eax    /* SID = 0xc00010ff */
11   je    func0_entry
12   hlt
13 func0_entry:
14   ...
15 func1:                       /* callee */
16   endbr64
17   sub   $0xbaddcafe, %eax    /* SID = 0xbaddcafe */
18   je    func1_entry
19   hlt
20 func1_entry:
21   ...
```

With FineIBT in place, before each indirect calls compiler instrument a code to move a random SID to any general purpose register (*eax* in above case) and on callee (*func0*), we verify if the value in *%eax* is matched or not using *sub* instruction at line 10. For all direct calls to any indirect function target, we create a clone of the original function (*func1_entry*) and call that rather than the original function.

Currently the technique is only supported in linux kernel.

Limitation of CFI – Forward edge

Over the years, numerous researchers surfaced that bypasses CFI completely or at a certain level. We will go through some well known cases below.

LLVM CFI limitations

Performance penalty:

Due to the inclusion of additional verification instructions and the incorporation of a runtime library, the performance of VTV can be affected, with a range of impact varying from 2% to 20% depending on how the application has been implemented i.e more virtual functions brings more performance impact.

The performance of IFCC is contingent upon the number of indirect calls executed by a program. In the majority of programs, the penalty incurred is less than 4%.

Limitation in capability:

The CFI present in CLANG is still not capable of protecting control flow divergence using CRA(Code reuse attack) based on backward edges i.e Return oriented programming.

> Note: Initial stage CFI implementation bypass research which is based on finding gadget on allowed targets using ROP:
> https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6956588

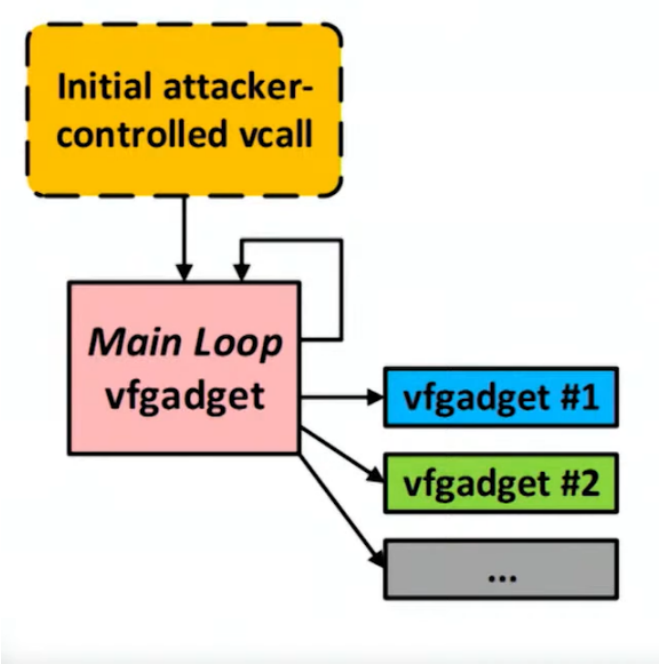
Code reuse attack for forward edge

Even though CFI's main goal is to protect Code reuse attack like ROP, there are certain type of CRA introduced over years to defeat existing CFI implementation. One such research mentioned below:

COOP - **Counterfeit Object-Oriented Programming** is a code reuse attack approach targeting applications developed in C++ or possibly other object-oriented languages. At high level, it relies on finding protected targets in the application binary which can legitimately be called and doesn't cause CFI violation.

COOP, virtual functions existing in an application are repeatedly invoked on counterfeit C++ objects carefully arranged by the attacker. Counterfeit objects are *not* created by the target application, but are injected in bulk by the attacker.

To understand in more details, COOP relies on existing virtual function reuse called "vfgadgets". Vfgadgets flow can be understood using below image:



Once an attacker is able to control the *vptr*, it will redirect the execution to Main loop *vfgadget* which executes in loop. From this main loop gadget, the attacker invoke the actual *vfgadget* that are injected on process memory as payload.

COOP can be use to bypass most CFI implementation besides LLVM or GCC VTV and SafeDispatch.

	Memory safety	Control-flow integrity	Code shuffling, rewriting, hiding	Heuristics
Binary code	• Cling [1] • ...	• CFI + shadow call stack [2] • CCFIR [3] • vfGuard [4]	• STIR [5] • Smashing The Gadgets [6]	• kBouncer [7] • EMET [8] • ROPecker [9]
Source code	• CPI [10] • CPS [10] • SoftBound [11] • WIT [12]	• GCC VTV [13] • LLVM IFCC [13] • SafeDispatch [14] • Windows CFG	• G-Free [15] • XnR [16] • Readactor [17]	

Image source: https://www.youtube.com/watch?v=NDt7Tholxp4

You can read more about the attack here: https://ieeexplore.ieee.org/document/7163058

Limitation in linux kernel:

For linux kernel the earlier default CFI implementation that was similar to CFI-CLANG was less powerful since even though the CFI reduce the attack surface to limited call sites, in linux kernel most function have prototype of *void foo(void)*

Limitation of CFG

Windows implementation of CFI named Control flow guard has two implementation limitations about requirement of ASLR and alignment of guard functions. If a binary doesn't support ASLR then CFG cannot be implemented in the following binary due to the fact that CFG relies on ASLR to work properly.

Besides that, CFG requires all guard functions to be aligned to 0x10. If the function call is not aligned to 0x10, it will use an odd bit only. This allows untrusted function call near trusted function call. In detail: CFG is able to precisely mark a valid target only if it is the only target in its address range and it is 16-byte aligned. In that case, the state will be 10. However,if a target is not aligned, or there are multiple targets in the same range, then the state will have to be set to 11, which allows branches to any address in the range. In other words,we can freely alter the lower 4 bits of a valid unaligned target and the result will still be a valid target. This enables us to reach code located near an unaligned function's entry point, which leads to interesting code sequences. You can read more about it here: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_05A-3_Biondo_paper.pdf

Unsupported module presence in process

CFG depends on compile and link level processing. As a result, third party modules and even old versions of MS binaries are not safeguarded by CFG. Furthermore, if the main executable image is not designed for CFG, CFG will be entirely disabled during the process, even if it loads system modules that do support CFG.

JIT code bypass

CFG doesn't support JIT generated code. It can contain unprotected code and all corresponding bits in the CFG Bitmap are set.

More details here:
https://www.blackhat.com/docs/us-15/materials/us-15-Zhang-Bypass-Control-Flow-Guard-Comprehensively-wp.pdf

CFI (Backward edge Integrity)

To ensure the effectiveness of CFI in various situations, hardware manufacturers have implemented several CFI backward edge techniques that closely resemble the workings of many first-generation techniques but rely heavily on hardware. However, the initial significant advancement in backward edge was introduced as a software solution in clang in 2014, which we will examine first.

SafeStack

The initial implementation of protection for backward edges was presented in a research paper published in 2014, which focused on Code Pointer Integrity (CPI). The paper also discussed SafeStack, a key element of Code Pointer Separation that provides defense for both return addresses and local variables. This protective measure was first introduced in clang 3.8 in the same year and continues to be utilized to this day.

Introduction to CPI

CPI fully protects the program against all control-flow hijack attacks that exploit program memory bugs. In a nutshell, it protect all types of code pointer (backward or forward edge) i.e it guarantees the integrity of all code pointers in a program (e.g., function pointers, saved return addresses) and thereby prevents all control-flow hijack attacks, including return-oriented programming.

The key idea behind CPI is to split process memory into a safe region and a regular region. CPI uses static analysis to identify the set of memory objects that must be protected in order to guarantee memory safety for code pointers. This set includes all memory objects that contain code pointers and all data pointers used to access code pointers indirectly. All objects in the set are then stored in the safe region, and the region is isolated from the rest of the address space (e.g., via hard-ware protection). The safe region can only be accessed via memory operations that are proven at compile time to be safe or that are safety-checked at runtime.

Safe Stack technical details

Safe stack is to protect the return address. It does that by placing all proven-safe objects(return address and local variables) onto a safe stack located in the safe region. The safe stack can be accessed without any checks.

The safe stack mechanism consists of a static analysis pass, an instrumentation pass, and runtime support. The analysis pass identifies, for every function, which objects in its stack frame are guaranteed to be accessed safely and can thus be placed on the safe stack; return addresses and spilled registers always satisfy this criterion. For the objects that do not satisfy this criterion, the instrumentation pass inserts code that allocates a stack frame for these objects on the regular stack.

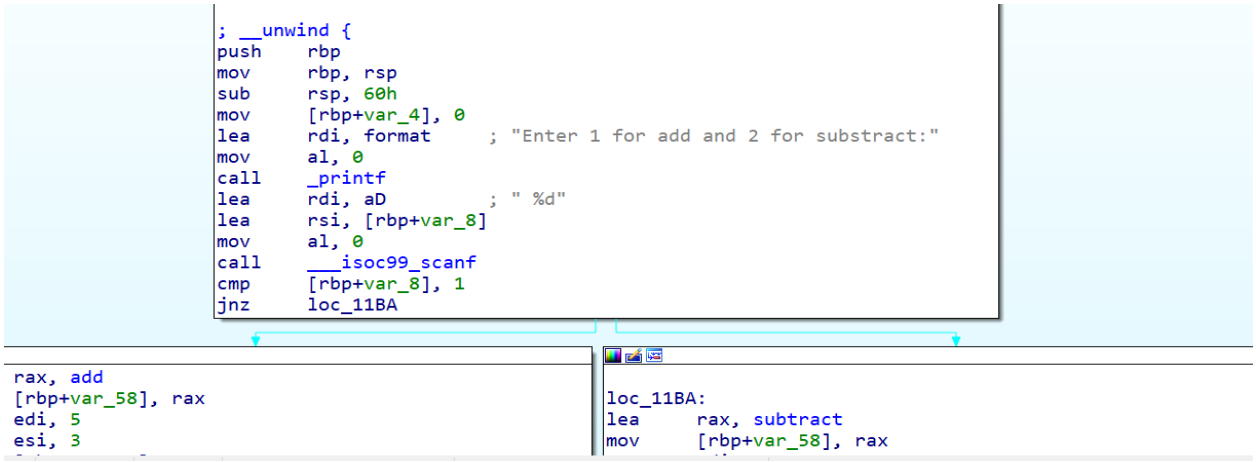
Safe Stack in action

Let's compile our above mentioned standard program with safe stack protection on. You can compile file with clang and pass *-fsanitize=safe-stack* flag.

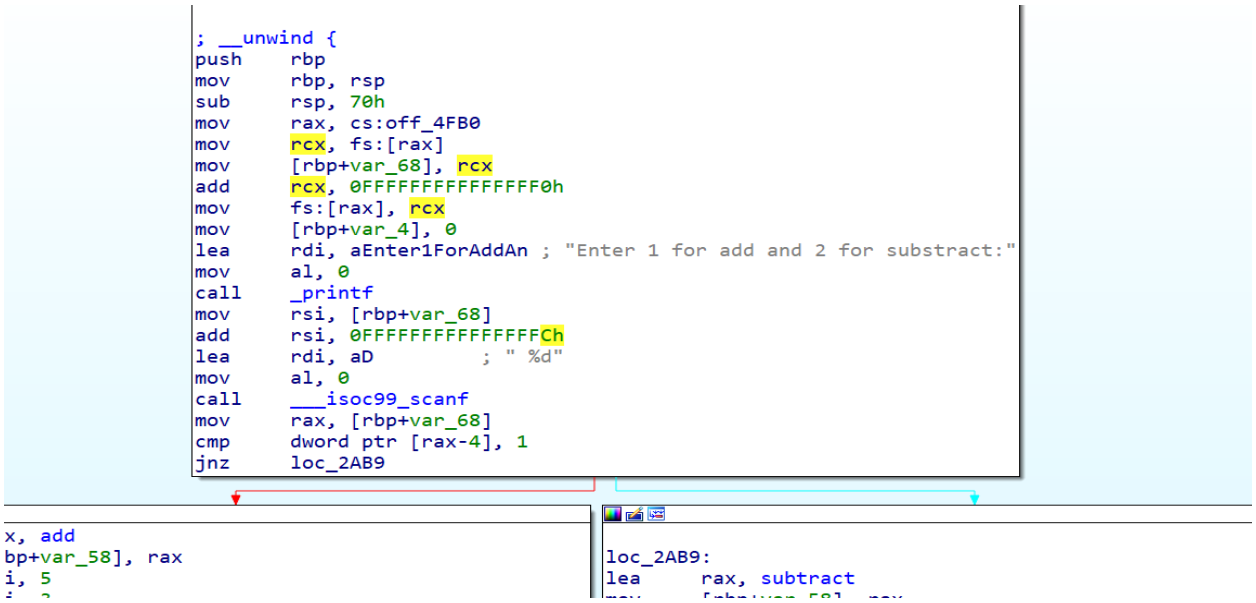
```
clang -fsanitize=safe-stack safestack.c -o safestack
```

You will see some instrumentation added to the program function.

Before:



After:



Resources:

You can read more about SafeStack and CPI here: https://dslab.epfl.ch/pubs/cpi.pdf.

TODO: Add safestack bypasses section

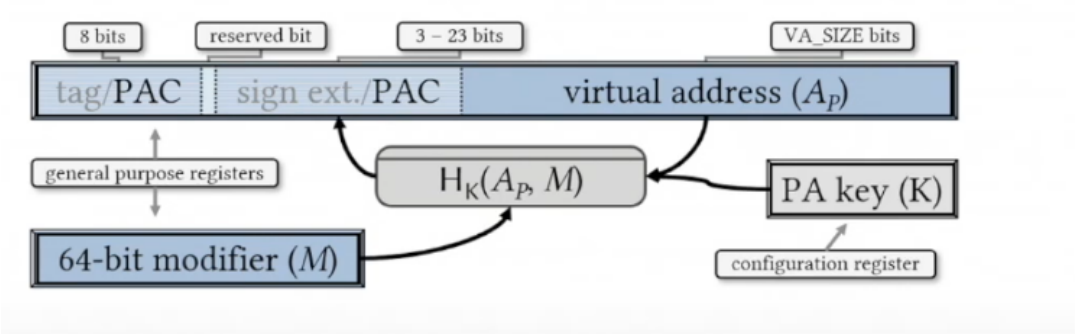
PAC (Pointer authentication code)

ARM has the distinction of introducing Pointer authentication, the first technique for backward edge, which was introduced in ARM v8.3 architecture that was released in late 2016. Subsequently, support for PA was added in gcc in 2017 (v7) and in the Linux kernel in 2018.

Pointer authentication not only focuses on protecting backward edges, but is also effective in scenarios involving modifications of all types of pointers, such as function or data pointer validations. However, it is most commonly used by compilers to protect backward edges, specifically return addresses.

Pointer authentication technical details

From the title, you can anticipate the purpose of pointer authentication, which is to verify whether a pointer is valid or not before utilizing it. ARM incorporates a PAC (Pointer Authentication Code) into every pointer that needs protection prior to storing it in memory, and confirms its integrity before using it. This PAC is stored in the top byte ignore bits (usually the 48th to 64th bit if Tagging is deactivated) of the virtual address space in ARM. In order to alter a protected pointer, an attacker would need to discover or guess the correct PAC in order to gain control over the program's flow.

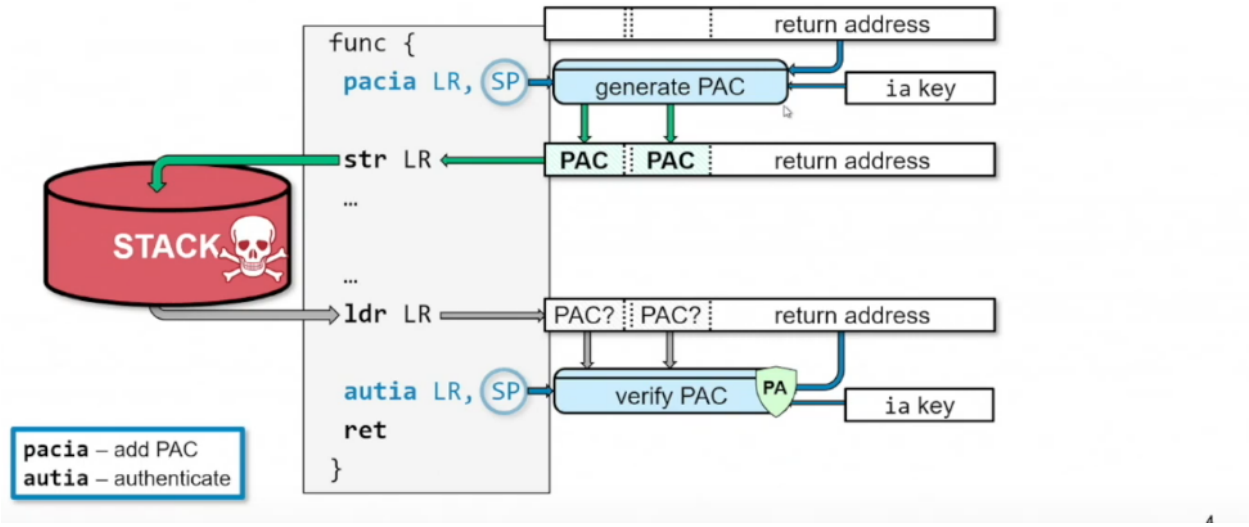


ARM uses a key generated for specific context to create PAC. The pointer authentication specification defines five keys: two for instruction pointers, two for data pointers and one for a separate general-purpose instruction for computing a MAC over longer sequences of data. The instruction encoding determines which key to use. For protection of key, it is stored in internal registers and are not accessible by EL0 (user mode), but otherwise are not tied to exception levels. Whenever a process is created, the kernel(running in EL1) will generate a random key and store it in that process's context; the process will then be able to use that key to sign and authenticate pointers, but it cannot read the key itself.

Generation and use of PAC is handled by two set of instructions: PAC* and AUT*. PAC* is used to compute and add PAC and AUT* is used for verifying the PAC. To generate PAC three values are used, the pointer itself, a secret key hidden in the process context, and a third value like the current stack pointer passed through a cipher called QARMA. PAC is the truncate output of the resulting cryptographic operation.

Implementation of PAC

PAC can be enabled in AARCH64 architecture using *CONTROL.PAC_EN* or *CONTROL.UPAC_EN* flags. The Pointer authentication flow can be understood using the diagram below.



Source: USENIX Security '19 - PAC it up: Towards Pointer Integrity using ARM Pointer Authentication

Pointer authentication in action:

Let's compile our above program with Pointer authentication on. You can pass one of the following flag to gcc:

```
-msign-return-address=all (deprecated)
```

or

```
-mbranch-protection=pac-ret
```

Let's check the changes of *main()* function due to PAC after compilation:

```
00000000000007d4 <main>:
 7d4:   d503233f        paciasp
 7d8:   a9bd7bfd        stp     x29, x30, [sp, #-48]!
 7dc:   910003fd        mov     x29, sp
 7e0:   90000000        adrp    x0, 0 <__abi_tag-0x278>
 7e4:   91238000        add     x0, x0, #0x8e0
 7e8:   97ffffaa        bl      690 <printf@plt>
 7ec:   910073e0        add     x0, sp, #0x1c
 7f0:   aa0003e1        mov     x1, x0
 7f4:   90000000        adrp    x0, 0 <__abi_tag-0x278>
 7f8:   91242000        add     x0, x0, #0x908
 7fc:   97ffffa1        bl      680 <__isoc99_scanf@plt>
 800:   b9401fe0        ldr     w0, [sp, #28]
 804:   7100041f        cmp     w0, #0x1
 808:   540001c1        b.ne    840 <main+0x6c>  // b.any
 80c:   90000000        adrp    x0, 0 <__abi_tag-0x278>
```

```
 844:   91228000        add     x0, x0, #0x8a0
 848:   f90017e0        str     x0, [sp, #40]
 84c:   f94017e2        ldr     x2, [sp, #40]
 850:   52800061        mov     w1, #0x3                        // #3
 854:   528000a0        mov     w0, #0x5                        // #5
 858:   d63f0040        blr     x2
 85c:   b90027e0        str     w0, [sp, #36]
 860:   b94027e1        ldr     w1, [sp, #36]
 864:   90000000        adrp    x0, 0 <__abi_tag-0x278>
 868:   9124c000        add     x0, x0, #0x930
 86c:   97ffff89        bl      690 <printf@plt>
 870:   52800000        mov     w0, #0x0                        // #0
 874:   a8c37bfd        ldp     x29, x30, [sp], #48
 878:   d50323bf        autiasp
 87c:   d65f03c0        ret
```

You will see *paciasp* instruction at top which will generate PAC and store it in stack and *rsp* register. At the epilog of the program, *autiasp* will verify if the value of PAC in stack is similar to what is present in address top bytes or not. On difference, the program will crash.

> Note: You will not see pointer authentication been used in add or subtract function since these function don't have local variables.

Resources

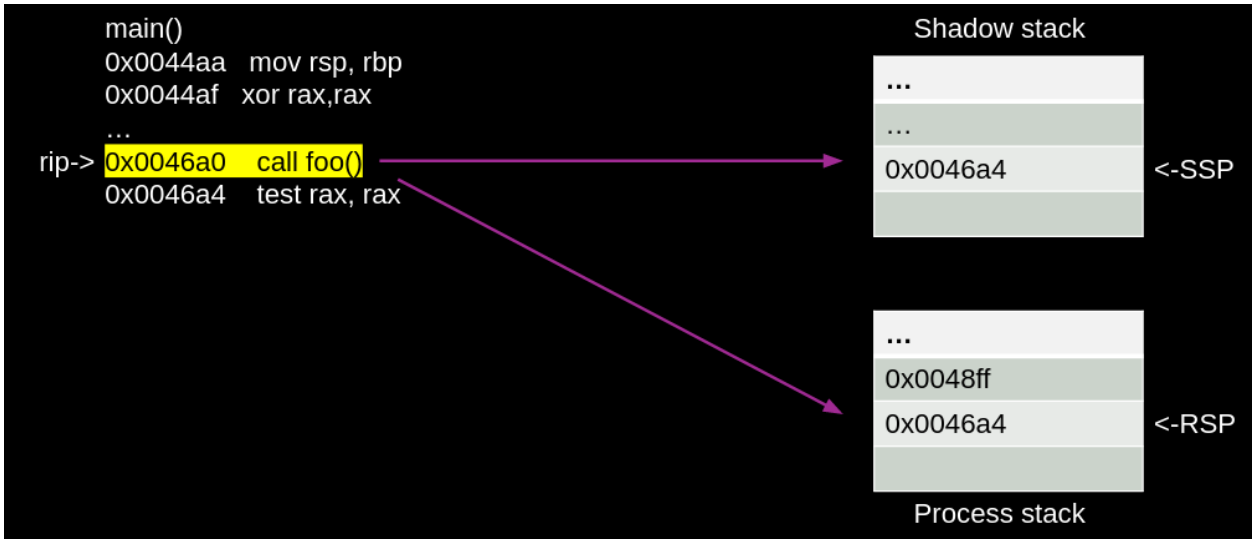
https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf

https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension

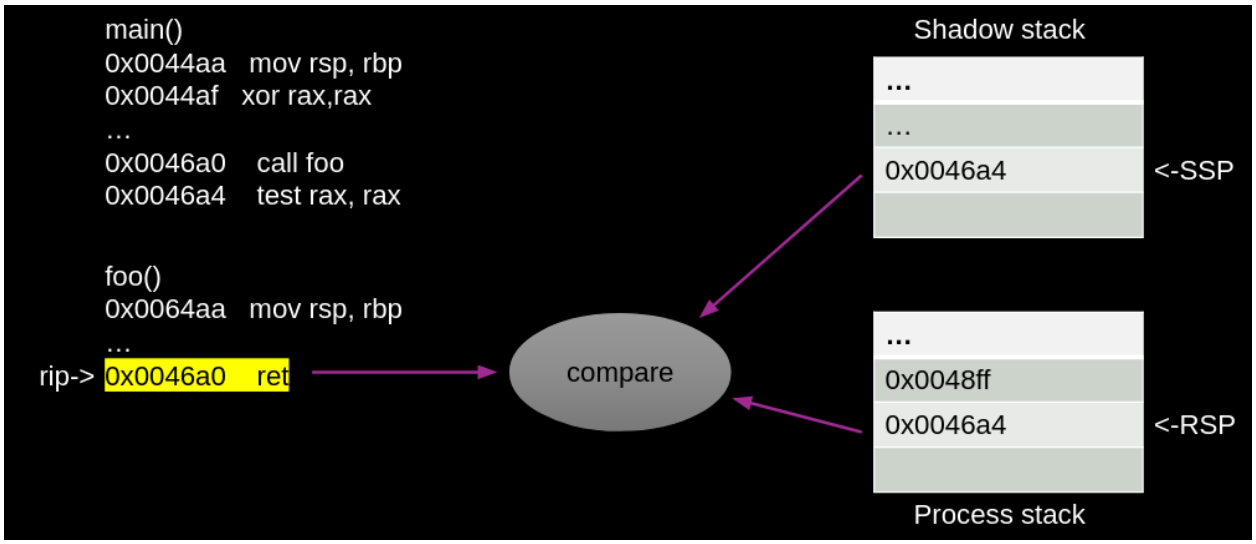
Shadow stack

As part of CET, intel has introduced shadow stack (along with IBT) from Intel Tigerlake processor released in 2020. Shadow stack is used to protect backward edge (i.e return address modification).

A shadow stack is a secondary stack allocated from memory which cannot be directly modified by the process. When shadow stacks are enabled, control transfer instructions/flows such as *near call, far call, call to interrupt/exception handlers*, etc. store their return addresses to the shadow stack and the process stack.



The *ret* instruction pops the return address from both stacks and compares them. In the event that the return addresses from the two stacks do not match, the processor will indicate a control protection exception (#CP).



The shadow stack is protected from tamper through the page table protections such that regular store instructions cannot modify the contents of the shadow stack. To provide this protection the page table

protections are extended to support an additional attribute for pages to mark them as “Shadow Stack” pages.

> Note: The idea of shadow stack originated from 2005 CFI research paper: https://dl.acm.org/doi/10.1145/1102120.1102165

Shadow stack in Linux - To compile binary with shadow stack support you can use -fcf-protection flag in both gcc and llvm. You will not see any instruction modification in CET compiled binary for shadow stack since it’s working is invisible from application.

Shadow stack in windows - In windows, you can use */CETCOMPAT* flag in visual studio 2019+ to compile binary with shadow stack support. You can read about shadow stack windows implementation from windows-internals blog https://windows-internals.com/cet-on-windows/.

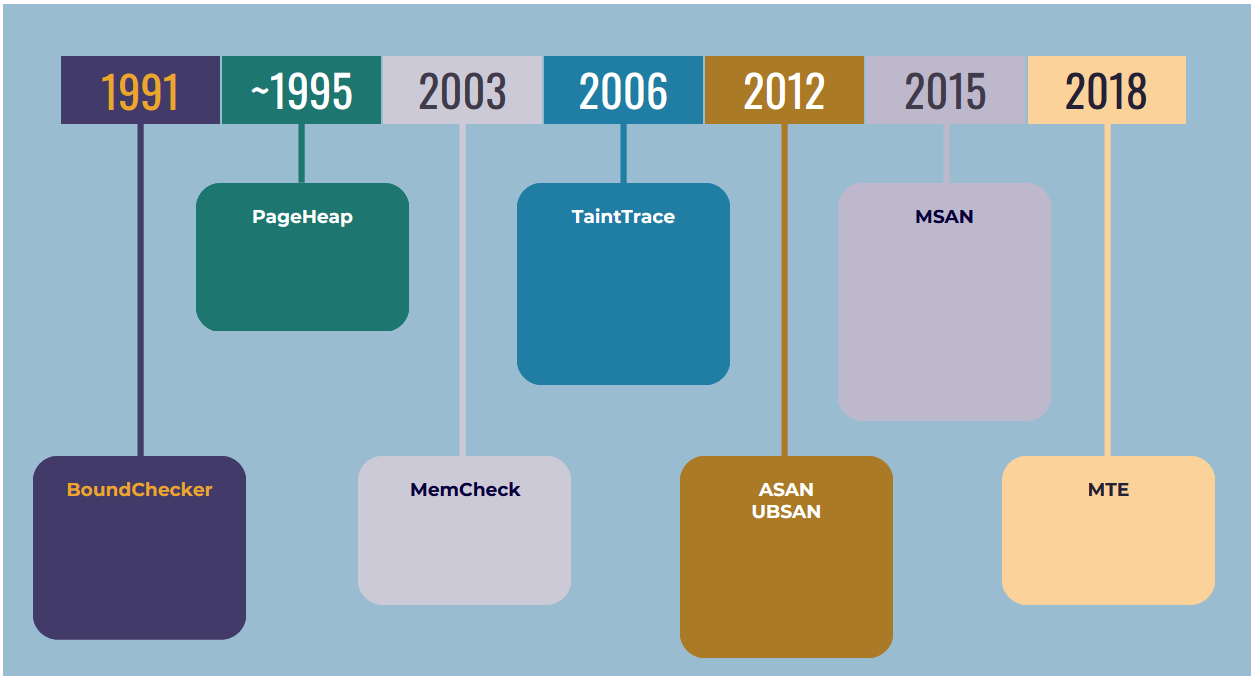
> Note: Before introduction of Intel CET, Windows implemented software based shadow stack technology called Return flow guard in Windows 10 Redstone 2 14942. You can read about RFG here: https://xlab.tencent.com/en/2016/11/02/return-flow-guard/

TODO: Add details on Backward edge limitations

.

Error detection tools (Heroes)

In this concluding chapter, we will explore tools (and few techniques) that have been specifically designed to improve the detection of memory errors in development or testing settings. While these tools often incur a significant performance overhead, they are not commonly utilized in production environments. However, they possess the full capacity to prevent memory corruption, similar to the techniques we have previously discussed in the first and second generation mitigations.



BoundsChecker

Around 1991 NuMega corp released a Memory leaks detection suite that is capable of detecting many kinds of array and buffer overrun conditions. Currently the tool is supported as part of Visual Studio in the form of Devpartner studio.

DevPartner suite has Error Detection functionality helps you find memory corruption problems caused by one of the following types of problems:

- Overrun allocated buffers (Buffer overflow)
- Continued access to memory after it has been deallocated (Use after free)
- Deallocating a resource multiple times (Double free)

The works by conducting instrumentation to effectively track memory usage and validate API calls. Unfortunately, the source code for this suite is not accessible, and due to its limited popularity, I made the decision to refrain from further exploration.

Limitations of BoundChecker

Due to the proprietary nature of Boundschecker, it has not been implemented anywhere and has not gained much popularity.

Because of its extensive instrumentation and runtime memory tracking, it is not utilized in production but rather serves as a testing suite during application development phases.

Furthermore, the tool has not been maintained for several years.

PageHeap

Sometime around 199X, Windows has added a feature of pageheap in WDK suite to monitor heap allocation and detect any access overruns in heap. PageHeap was present in GFlags, the Global Flags Editor, that is use to enable and disable advanced debugging, diagnostic, and troubleshooting features.

PageHeap has a very straightforward implementation. When a process is started with GFlag, each heap allocationcan be either writes fill patterns at the end of each heap allocation and examines the patterns when the allocations are freed, or it places an inaccessible page at the end of each allocation so that the program stops immediately if it accesses memory beyond the allocation.

Using PageHeap:

- To enable standard page heap verification for all processes, use *gflags /r +hpa* or *gflags /k +hpa*.
- To enable standard page heap verification for one process, use *gflags /p /enable ImageFileName*.
- To enable full page heap verification for one process, use *gflags /i ImageFileName +hpa* or gflags /p /enable ImageFileName /full.

TODO: Add PageHeap in action section

Memcheck (Valgrind)

Memcheck is a module in Valgrind project added in 2003. The whole Valgrind project uses dynamic memory instrumentation to work, so does memcheck.By employing memcheck, all memory reads and writes undergo thorough examination, and calls to malloc/new/free/delete are closely monitored. Consequently, Memcheck has the capability to identify various issues:

- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks - where pointers to malloc'd blocks are lost forever
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Overlapping src and dst pointers in memcpy() and related functions

Technical details

Memcheck performs four kinds of memory error checking.

- First, it tracks the addressability of every byte of memory, updating the information as memory is allocated and freed. With this information, it can detect all accesses to unaddressable memory.

- Second, it tracks all heap blocks allocated with *malloc*(), new and new[]. With this information it can detect bad or repeated frees of heap blocks, and can detect memory leaks at program termination.
- Third, it checks that memory blocks supplied as arguments to functions like *strcpy*() and *memcpy*() do not overlap. This does not require any additional state to be tracked.
- Fourth, it performs definedness checking: it tracks the definedness of every bit of data in registers and memory. With this information it can detect undefined value errors with bit precision.

Memcheck uses something called shadow memory to keep track of addressability of process original memory. The Valgrind framework intercepts function and system calls which cause usable address ranges to appear/disappear. Memcheck is notified of such events and marks shadow memory appropriately. For example, malloc and mmap bring new addresses into play: *mmap* makes memory addressable and defined, whilst malloc makes memory addressable but undefined. Similarly, whenever the stack grows, the newly exposed area is marked as addressable but undefined. Whenever memory is deallocated, the deallocated area also has its values all marked as undefined. Memcheck also uses such events to update its maps of which address ranges are legitimately addressable. By doing that it can detect accesses to invalid addresses, and so report to the user problems such as buffer overruns, use of freed memory, and accesses below the stack pointer.

Moreover, It normally uses 2 bits of shadow memory(explained in detail in next section) per byte of application memory; the shadow for every byte has 4 states: addressable and initialized, not addressable, address-able but uninitialized, addressable and partially initialized. If the byte is partially initialized then the tool maintains a second layer of shadow, this time with bit-to-bit mapping.

The most complex part of Memcheck working is the definedness feature. The basic idea underlying the definedness checking is as follows.

• Every single bit of data, b, maintained by a program, in both registers and memory, is shadowed

The first three checks are done using instrumentation by a piece of metadata, called a definedness bit. For historical reasons these are often also referred to as V bits (V being short for “validity”). Each V bit indicates whether or not the bit shadows is regarded as currently having a properly defined value.

Every single operation that creates a value is shadowed by a shadow operation that computes the V bits of any outputs, based on the V bits of all inputs and the operation. The exact operations performed by this shadow computation are important, as they must be sufficiently fast to be practical, and sufficiently accurate to not cause many false positives.

• Every operation that uses a value in such a way that it could affect the observable behavior of a program is checked. If the V bits indicate that any of the operation’s inputs are undefined, an error message is issued. The V bits are used to detect if any of the following depend on undefined values: control flow transfers, conditional moves, addresses used in memory accesses, and data passed to system calls.

A V bit of zero indicates that the corresponding data bit has a properly defined value, and a V bit of one indicates that it does not. Every 32-bit general purpose register is shadowed by a 32-bit shadow register, and every byte of memory has a shadow V byte.

Memcheck in action

Let’s run the classic_overflow program under with valgrind:

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
        char password[64];
        strcpy(password, argv[1]);
        if (strcmp(password, "secret") == 0)
        {
                printf("Sucessfully login\n");
        }
        else
        {
                printf("Password doesn't match. Unanble to login.\n");
        }
        return 0;
}
```

Output:

```
shubham@MININT-1T2PIDD:~/asan$ valgrind --tool=memcheck ./classic_overflow aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
==452== Memcheck, a memory error detector
==452== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==452== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==452== Command: ./classic_overflow aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaa
==452==
Password doesn't match. Unanble to login.
==452== Jump to the invalid address stated on the next line
==452==    at 0x6161616161616161: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x60: ???
==452==  Address 0x6161616161616161 is not stack'd, malloc'd or (recently) free'd
==452==
==452==
==452== Process terminating with default action of signal 11 (SIGSEGV)
==452==  Bad permissions for mapped region at address 0x6161616161616161
==452==    at 0x6161616161616161: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x6161616161616160: ???
==452==    by 0x60: ???
==452==
==452== HEAP SUMMARY:
==452==     in use at exit: 0 bytes in 0 blocks
==452==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==452==
==452== All heap blocks were freed -- no leaks are possible
==452==
==452== For lists of detected and suppressed errors, rerun with: -s
==452== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault
```

In the output, you will observe that memcheck is able to detect an overflow scenario and display the stack data for the purpose of debugging.

Limitations of MemCheck

Due to the definedness feature and extensive instrumentation, running the program with memcheck can result in a slowdown of up to 40%. Additionally, the memory usage is doubled due to the inclusion of shadow bytes. As a result, memcheck is typically only used in a testing environment to identify bugs in the application. These tools are commonly employed to either track down specific bugs or confirm the absence of any hidden bugs (which can be detected by Memcheck) in the code.

In addition to the performance penalty, an important limitation of Memcheck is its inability to detect all cases of bounds errors in the use of static or stack-allocated data. The following code will pass the Memcheck tool in Valgrind without incident, despite containing the errors described in the comments:

```
int Static[5];

  int func(void)
  {
    int Stack[5];

    Static[5] = 0;  /* Error - Static[0] to Static[4] exist, Static[5] is out of
bounds */
    Stack [5] = 0;  /* Error - Stack[0] to Stack[4] exist, Stack[5] is out of
bounds */

    return 0;
  }
```

Resources:

https://nnethercote.github.io/pubs/memcheck2005.pdf

https://www.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-40/Nice/RuleRefinement/bin/valgrind-3.2.0/docs/html/mc-manual.html

Taint trace

Developed in 2006 by researcher from MIT, Tainttrace was a tracing tool that protects systems from software exploits. The tool is capable of protecting against following types of memory corruptions:

- Buffer overflows
- Format string attacks
- Indirect branch modifications

Tainttrace worked dynamically hence doesn't need any specific compilation. It uses DynamoRio for dynamic instrumentation. Besides that, it uses 1to1 shadow memory mapping for application memory to keep track of memory structure of process. More information below from there research paper:

> The system consists of four components. A configuration file is used to specify the security policy. The shadow memory is a data structure used to maintain the taint information of application data. Program monitor is the core module used to perform the instrumentation, intercept system calls, and enforce security policies. A customized loader is used to load the application binary, shadow memory, and program monitor into different memory spaces. To start an application, our loader first loads the various components into specific memory spaces and then passes control to the program monitor. The program monitor reads the configuration file and sets up the tracing policy. It also initializes the shadow memory, that is, it marks the un-trusted sources specified by the configuration file as tainted, and other sources as clear. After initialization, the application executes under our program monitor. All the code to be executed in user mode is first copied into the code cache. This includes application code and shared libraries. The program monitor inserts additional code for maintaining, propagating, and checking taint status before executing the code. In this way, we achieve comprehensive information flow tracing. At crit- ical program points specified by our policy (e.g. indirect branch), run-time condition checking is performed to restrict sensitive data usage.

Shadow memory

As mentioned above, Tainttrace uses shadow memory to keep track of which memory is tainted and which in not. It uses following offset to store the shadow bytes representing the original program bytes:

l2 = l1[(addr >> 16) & 0xffff];

shadow = &l2[addr & 0xffff];

It uses a simple addressing strategy that maps the shadow memory byte by adding a constant offset, shadow base, to the application memory byte address. The customized loader partitions the memory space to support this mapping. This byte- to-byte mapping makes taint propagation simple and efficient.

0 – Good byte

1 – Bad byte

Implementation details

The loader is implemented by modifying the source code of Valgrind. It consists of two stages. In stage 1, it loads the code of stage 2 into the monitor space (*0xb0000000 to 0xbfffffff*) and transfers control to stage 2. In stage 2, the application and its shared libraries are loaded into the application space *(0x000000000 to 0x57f00000*). It also loads DynamoRIO into the monitor space and transfers control to DynamoRIO. DynamoRIO loads our program monitor, dr-instrument.so, implemented as a shared library, into the monitor space. DynamoRIO constructs basic blocks for execution and instrument. so is used to perform the instrumentation and intercept system calls. Syscall interception is used for several purposes:

allocating shadow memory, marking taint status for data read from files or sockets, and modifying temporary file operations.

Limitations of TaintTrace

The main problem with TaintTrace is its performance overhead of 5 times, which is caused by the need for instrumentation and continuous memory monitoring. Additionally, it should be noted that it requires twice the amount of memory compared to the default due to the use of shadow memory.

Since its release in 2006, the developer has not actively maintained it for several years, making it largely deprecated and not widely used.

Resources:

https://wiki.aalto.fi/download/attachments/65019433/Jukka_Julku_dynamic_program_analysis_tools_for_software_security.pdf?version=1&modificationDate=1336810687000&api=v2

ASAN (AddressSanatizer)

Address Sanatizer (ASAN) was first introduced by Google in 2012. Unlike first and second gen mitigations, this was introduced to detect memory corruption bugs in the debug environment and never meant to be part of the production system due to its performance implications.

In summary, AddressSanitizer (aka ASan) is a memory error detector tool that is implemented in different compiler like gcc, clang that detects bugs in the form of undefined or suspicious behavior by a compiler inserting instrumentation code at runtime. Asan is capable of detecting following class of memory corruption bugs:

- Use after free (dangling pointer dereference)
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

Address Sanitizer Implementation details:

AddressSanitizer consists of two parts: an instrumentation module and a run-time library. The instrumentation module modifies the code to check the shadow state for each memory access and creates poisoned red-zones around stack and global objects to detect overflows and underflows. The run-time library replaces malloc, free and related functions, creates poisoned redzones around allocated heap regions, delays the reuse of freed heap regions, and does error reporting.

Let's try to run our classic buffer overflow program and observe the output produced by ASAN.

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
        char password[64];
        strcpy(password, argv[1]);
        if (strcmp(password, "secret") == 0)
        {
                printf("Sucessfully login\n");
        }
        else
        {
                printf("Password doesn't match. Unanble to login.\n");
        }
        return 0;
}
```

You can compile it using gcc and clang since both support ASAN, while compiling provide a special flag to tell the compiler to build binary with ASAN support.

```
clang -O1 -g -fsanitize=address -fno-omit-frame-pointer classic_overflow.c
```

Let’s execute the above program with a buffer overflow scenario:

```
shubham@MININT-1T2PIDD:~/asan$ ./a.out aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
=================================================================
==13==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffcd496dcc0 at pc 0x0000004840d8 bp 0x7ffcd496dc50 sp 0x7ffcd496d410
WRITE of size 128 at 0x7ffcd496dcc0 thread T0
    #0 0x4840d7 in strcpy (/home/shubham/asan/a.out+0x4840d7)
    #1 0x4c8015 in main /home/shubham/asan/classic_overflow.c:7:2
    #2 0x7fd8e0518d09 in __libc_start_main csu/../csu/libc-start.c:308:16
    #3 0x41e2b9 in _start (/home/shubham/asan/a.out+0x41e2b9)

Address 0x7ffcd496dcc0 is located in stack of thread T0 at offset 96 in frame
    #0 0x4c7f5f in main /home/shubham/asan/classic_overflow.c:5

  This frame has 1 object(s):
    [32, 96) 'password' (line 6) <== Memory access at offset 96 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow (/home/shubham/asan/a.out+0x4840d7) in strcpy
Shadow bytes around the buggy address:
  0x10001a925b40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001a925b50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001a925b60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001a925b70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001a925b80: 00 00 00 00 00 00 00 00 00 00 00 00 f1 f1 f1 f1
=>0x10001a925b90: 00 00 00 00 00 00 00 00[f3]f3 f3 f3 00 00 00 00
  0x10001a925ba0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001a925bb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001a925bc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001a925bd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001a925be0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
```

In the above scenario, it is evident that ASAN is capable of detecting the buffer overflow. Subsequently, the output provides the call stack. Adjacent to the call stack is the line number and the event that triggers the overflow. Following that, we possess the shadow memory that ASAN has preserved for the process. Let us delve deeper into the significance of this shadow memory.

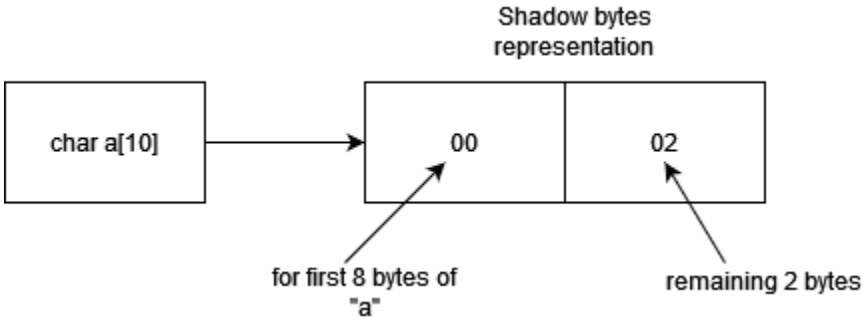
Shadow memory

ASAN maintains a separate memory for itself in the process memory that keeps track of the original process memory (i.e stack and heap allocations). This shadow memory has information about the current state of the original memory. ASAN allocates 1 byte of shadow memory to each 8 bytes of application/process memory. These bytes can have one of the values:

- FX (F1 to FF) – To refer the red zones. Red zones can also be referred as guard bytes that are memory that is not addressable to application flow i.e process events should not read/write these bytes (similar to guard pages).
- 00 – Complete 8 bytes that this shadow byte point is addressable.
- 0X (X lies between 0-7) – X number of bytes are addressable.

Let's try to understand it with an example.

An allocation of 10 bytes using *char a[10]* will be represented as *00 02*. Where 00 represents the first 8 bytes of allocated memory for "a" and 02 for the remaining 2 bytes.



Few more examples below:

Process bytes	Associated shadow bytes
f1 4a 34 65 2d 43 11 11	00
f1 4a 34 65 2d 43 11 11 45 23	00 02
char a[12]	00 04
Int a[8]	00 00 00 00 00 00 00 00
char a[2]	02

Each of these shadow bytes are presented at fix offset to the original byte location: The ASAN runtime calculates the shadow byte address using following *formula -> (address >> 3) + some_offset*. This offset value is defined or set by the compiler during compilation.

Instrumentation:

Address sanitizer uses instrumentation to add the red zones around the boundary of each allocation and verify the shadow bytes on every access. When instrumenting an 8-byte memory access, Address-Sanitizer computes the address of the corresponding shadow byte, loads that byte, and checks whether it is zero:

```
ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0)
ReportAndCrash(Addr);
```

Runtime Library

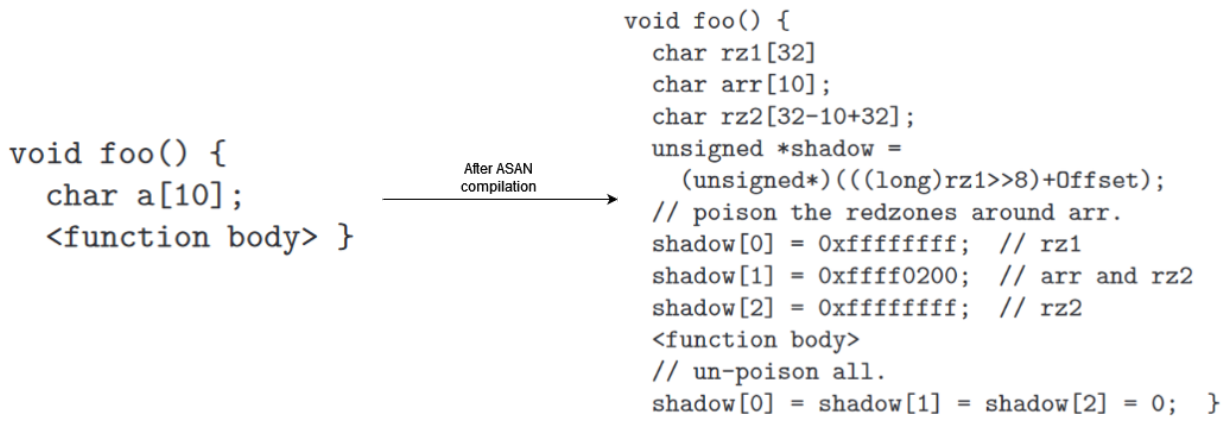
The main purpose of the run-time library is to manage the shadow memory. At application startup the entire shadow region is mapped so that no other part of the program can use it. For memory corruption detection against heap, malloc and free functions are replaced with a specialized implementation. The malloc function allocates extra memory, the redzone, around the returned region. The redzones are marked as unaddressable, or poisoned. The memory regions inside the allocator are organized as an array of freelists corresponding to a range of object sizes. When a freelist that corresponds to a requested object size is empty, a large group of memory regions with their redzones is allocated from the operating system. Each redzones is of minimum 32 bytes in size and looks something like this in memory

RedZone 1	Memory 1	RedZone 2	Memory 2	RedZone 3

The free function poisons the entire memory region and puts it into quarantine, such that this region will not be allocated by malloc any time soon.

For globals, the redzones are created at compile time and the addresses of the redzones are passed to the run- time library at application startup.

For stack objects, the redzones are created and poisoned at run-time. Currently, redzones of 32 bytes (plus up to 31 bytes for alignment) are used.



ASAN in action

Let's try to understand the instrumentation done by ASAN using a simple C program:

```c
#include <stdio.h>
#include <string.h>
int main()
{
        char a[20];
        strcpy(a, "hello");
        printf ("The string is %s\n",a);
        return 0;
}
```

Before ASAN:

```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_18= dword ptr -18h
var_14= word ptr -14h
a= byte ptr  8

; __unwind {
sub     rsp, 18h
mov     [rsp+18h+var_14], 6Fh ; 'o'
mov     [rsp+18h+var_18], 6C6C6568h
mov     rsi, rsp
mov     edi, offset format ; "The string is %s\n"
xor     eax, eax
call    _printf
xor     eax, eax
add     rsp, 18h
retn
; } // starts at 401130
main endp
```

After ASAN:

```
mov     [rbx+18h], r12
lea     r15, [r12+20h]
mov     qword ptr [r12], 41B58AB3h
mov     qword ptr [r12+8], offset a132203A5 ; "1 32 20 3 a:5"
mov     qword ptr [r12+10h], offset main
mov     r13, r12
shr     r13, 3
mov     rax, 0F3F8F8F8F1F1F1F1h
mov     [r13+7FFF8000h], rax
mov     dword ptr [r13+7FFF8008h], 0F3F3F3F3h
mov     word ptr [r13+7FFF8004h], 0
mov     byte ptr [r13+7FFF8006h], 4
mov     esi, offset _str ; "hello"
mov     edx, 6          ; void *
mov     rdi, r15        ; this
call    __asan_memcpy
mov     edi, offset _str_1 ; "The string is %s\n"
mov     rsi, r15
xor     eax, eax
call    printf
mov     word ptr [r13+7FFF8004h], 0F8F8h
mov     byte ptr [r13+7FFF8006h], 0F8h
mov     qword ptr [r12], 45E0360Eh
test    r14, r14
```

Below instructions are to create a fake stack frame for ASAN

```
mov     [rbx+18h], r12
lea     r15, [r12+20h]
mov     qword ptr [r12], 41B58AB3h
mov     qword ptr [r12+8], offset a132203A5 ; "1 32 20 3 a:5"
mov     qword ptr [r12+10h], offset main
```

Below instructions is to reach to the address of shadow memory using (address >> 3). This offset can be changed based on ASAN flags.

```
mov     qword ptr [r12+10h], offset main
mov     r13, r12
shr     r13, 3
```

If you look at the snippet below, the F1 bytes moved to *r13+7FFF8000* is the stack left redzone and bytes at *r13+7FFF8008* is the right redzone. From *r13+7fff8004* to *7fff8008* is the representation of our buffer *a[20]* which is represented as 00 00 04 (equivalent to 20 bytes).

```
mov     rax, 0F3F8F8F8F1F1F1F1h
mov     [r13+7FFF8000h], rax
mov     dword ptr [r13+7FFF8008h], 0F3F3F3F3h
mov     word ptr [r13+7FFF8004h], 0
mov     byte ptr [r13+7FFF8006h], 4
```

Full list of redzone bytes identification below:

```
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
```

You will also observe that the *memcpy* has been substituted with *asan_memcpy*, which is a component of the ASAN runtime library employed to ascertain if we are copying the data to the accurate location and no redzone bytes are being written. A very comparable ASAN concept operates for Heap.

Currently ASAN is supported in following compilers:

- Clang (starting from version 3.1)
- GCC (starting from version 4.8)
- Xcode (starting from version 7.0)
- MSVC (widely available starting from version 16.9).

Limitation of ASAN

1. On an average, the ASAN instrumentation enhances the processing time by approximately 73% and the memory usage by 240%. Due to this reason, it is never utilized in production but instead confined to usage in a debugging environment or utilized by fuzzers such as libfuzzer and AFL to identify memory corruptions.
2. TheASAN instrumentation may miss a very rare type of bug: an unaligned access that is partially out-of-bounds. For example:

```
int *a = new int[2]; // 8-aligned
```

```
 int *u = (int*)((char*)a + 6);
*u = 1; // Access to range [6-9]
```

if an out-of-bounds access touches memory too far away from the object bound it may land in a different valid allocation and the bug will be missed.

```
char *a = new char[100];
char *b = new char[1000];
a[500] = 0; // may end up somewhere in b
```

Resources:

https://storage.googleapis.com/pub-tools-public-publication-data/pdf/37752.pdf

UBSAN (UndefinedBehaviorSanitizer)

UBSAN was introduced in 2012 in clang project starting from version 3.3 and 2013 in GCC since version 4.9. UBSAN uses compile-time instrumentation to catch undefined behavior during program execution.

These are some common types of bugs that UBSAN detect:

- Array subscript out of bounds, where the bounds can be statically determined
- Bitwise shifts that are out of bounds for their data type
- Dereferencing misaligned or null pointers
- Signed integer overflow
- Conversion to, from, or between floating-point types which would overflow the destination

It works in similar fashion like any other sanitizers, by instrumenting every memory load.

UBSAN in action

Let's look at the case of null pointer dereference in C program:

```
#include <stdio.h>
int main() {
    int a, c; // some integers
    int *pi;     // a pointer to an integer
    a = 5;
    pi = NULL;
    c = *pi; // this is a NULL pointer dereference

}
```

Compile it using clang and execute it:

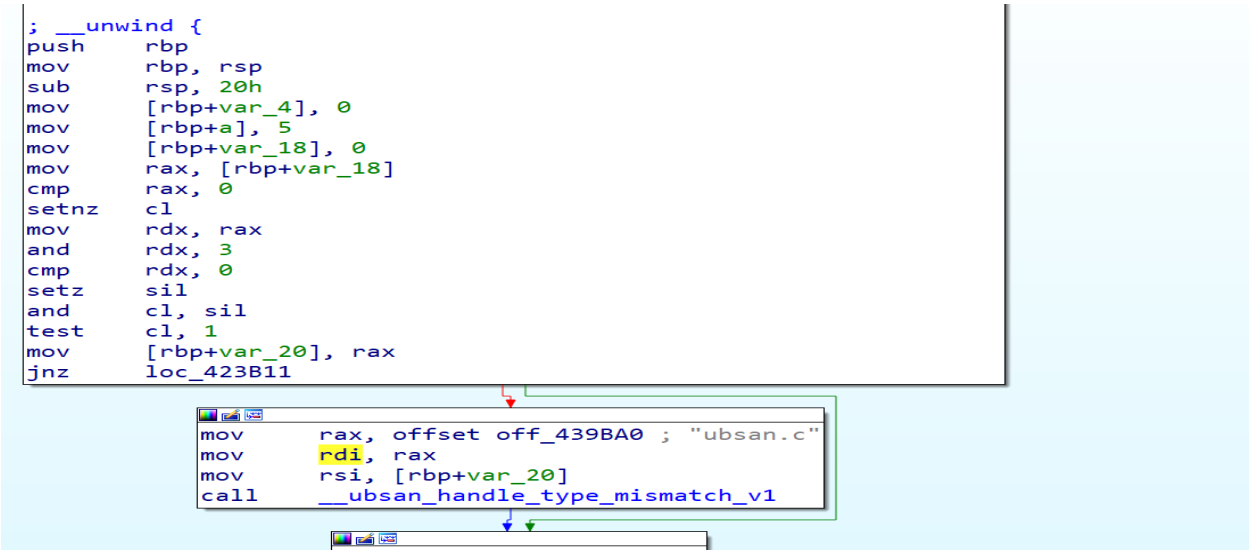
```
shubham@MININT-1T2PIDD:~/ubsan$ clang -fsanitize=undefined ubsan.c
shubham@MININT-1T2PIDD:~/ubsan$ ./a.out
ubsan.c:11:9: runtime error: load of null pointer of type 'int'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior ubsan.c:11:9 in
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==53==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address 0x000000000000 (pc 0x000000423b15 bp 0x7ffd5f786df0 sp
0x7ffd5f786dd0 T53)
==53==The signal is caused by a READ memory access.
==53==Hint: address points to the zero page.
    #0 0x423b15 in main (/home/shubham/ubsan/a.out+0x423b15)
    #1 0x7f20282b0d09 in __libc_start_main csu/../csu/libc-start.c:308:16
    #2 0x4032f9 in _start (/home/shubham/ubsan/a.out+0x4032f9)

UndefinedBehaviorSanitizer can not provide additional info.
SUMMARY: UndefinedBehaviorSanitizer: SEGV (/home/shubham/ubsan/a.out+0x423b15) in main
==53==ABORTING
```

You can see UBSAN has detected null pointer dereference occur due to *c=*pi*. Let's see how the compiled binary code changes.

Before

```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_10= qword ptr -10h
var_8= dword ptr -8
var_4= dword ptr -4

; __unwind {
push    rbp
mov     rbp, rsp
xor     eax, eax
mov     [rbp+var_4], 5
mov     [rbp+var_10], 0
mov     rcx, [rbp+var_10]
mov     edx, [rcx]
mov     [rbp+var_8], edx
pop     rbp
retn
; } // starts at 401110
main endp
```

After



You will notice in UBSAN compiled code adds null pointer check after each memory load into register using *cmp* instruction. At the end it has *__ubsan_handle_type_mismatch* which detects any NULL pointer access, unaligned memory access, or accessing memory from a pointer whose data is an insufficient size.

Limitations of UBSAN

Similar to ASAN, UBsan has performance related drawback. Adding UBSan instrumentation slows down programs by around 2 to 3x and increases the file size by around 20 times.

Resources:

https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

https://blogs.oracle.com/linux/post/improving-application-security-with-undefinedbehaviorsanitizer-ubsan-and-gcc

MSAN (Memory Sanitizer)

The MSAN was initially presented in 2015 by the same group that was accountable for creating the ASAN. The primary objective of the MSAN was to identify uninitialized memory in the C/C++. In a manner similar to the ASAN, it operates through compile time instrumentation, but at present, it is only compatible with CLANG.

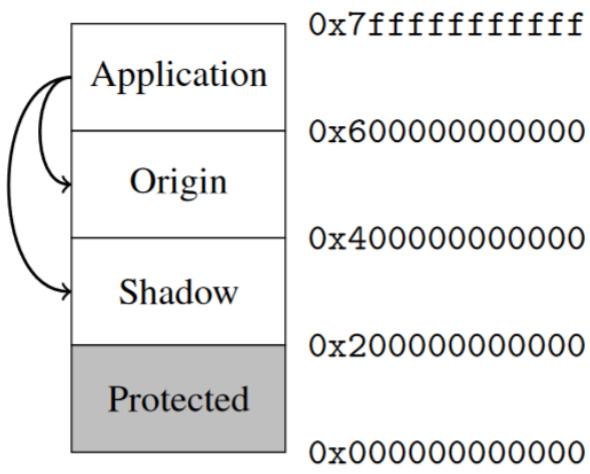
MSAN implementation details

MSAN uses shadow memory, instrumentation and runtime library similar to ASAN but doesn't require red zones to work. Detailed working explained below.

Shadow memory

MemorySanitizer employs 1-to-1 shadow mapping, i.e. for each bit of application memory the tool keeps one bit of shadow memory. Bit 0 in shadow memory stands for initialized, or defined bit, and value 1 — for uninitialized (undefined) bit. All newly allocated memory is "poisoned", i.e. corresponding shadow memory is filled with *0xFF*, signifying that all bits of this memory are uninitialized.

For origin tracking it allocates another region of the same size immediately following the shadow memory region.



Instrumentation

MemorySanitizer needs to handle all possible LLVM IR (SSA-based program representation) instructions either by checking operand shadow, or by propagating it to the result shadow. For every IR temporary value MemorySanitizer creates another temporary that holds its shadow value.

Runtime library

MemorySanitizer runtime library shares much common code with AddressSanitizer and ThreadSanitizer libraries. At startup it makes the lower protected area inaccessible, and maps Shadow and, optionally, Origin areas. MemorySanitizer uses the same allocator as the other Sanitizer tools. It does not add redzones around memory allocations, and does not implement memory quarantine. Allocated regions (with the exception of calloc regions) are marked as uninitialized, or 'poisoned'. Deallocated regions are marked uninitialized as well.

MemorySanitizer also implements origin tracking, which helps users to understand the errors. In origin tracking mode, MemorySanitizer associates a 32-bit origin value with each application value. This value serves as an identifier of a memory allocation (either heap or stack) that created the uninitialized bits this value depends on.

MSAN in action

Let's look at MSAN in action with a small illustration.

```
#include <stdio.h>
int main(int argc, char** argv) {
  int a[10];
  a[5] = 0;
  if (a[argc])
    printf("xx\n");
  return 0;
}
```

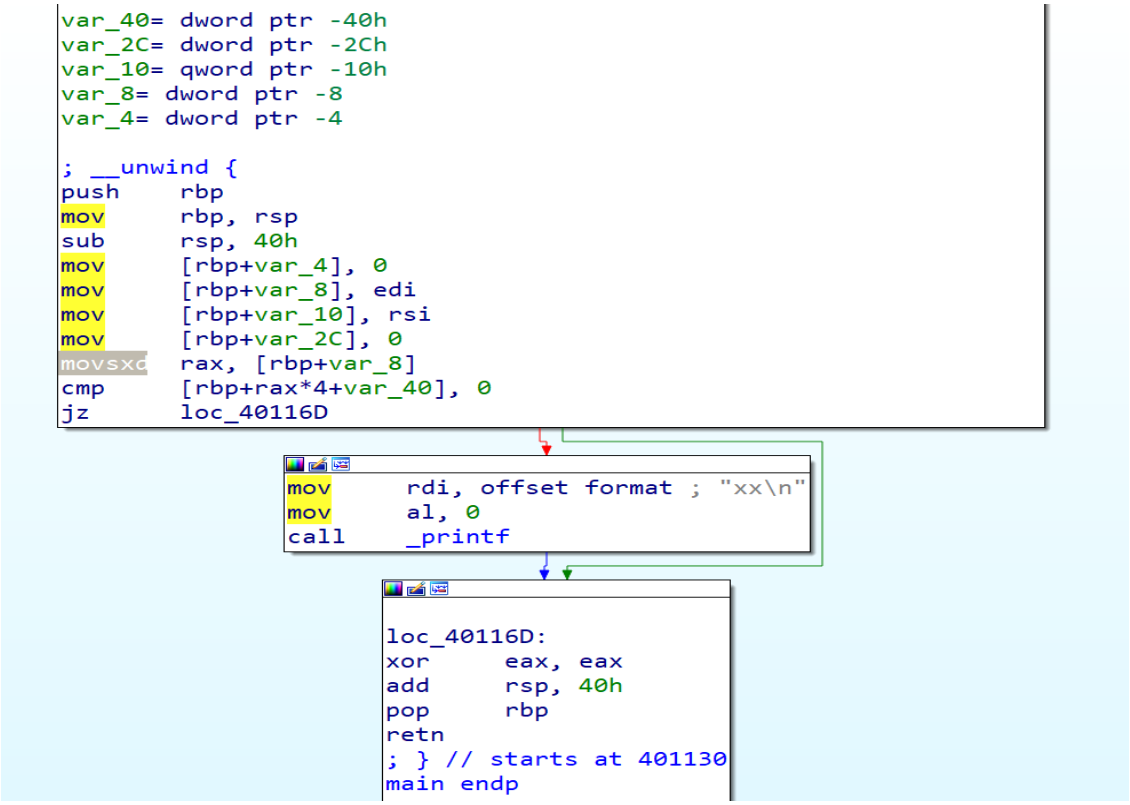
Compile the above program with MSAN and execute it. You will find following report from MSAN

```
shubham@MININT-1T2PIDD:~/msan$ clang -fsanitize=memory -fPIE -pie -fno-omit-frame-pointer -g -O2 msan_test.c
shubham@MININT-1T2PIDD:~/msan$ ./a.out
==34==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x55ac03f6a947 in main /home/shubham/msan/msan_test.c:8:7
    #1 0x7f9b0875dd09 in __libc_start_main csu/../csu/libc-start.c:308:16
    #2 0x55ac03ef0249 in _start (/home/shubham/msan/a.out+0x20249)

SUMMARY: MemorySanitizer: use-of-uninitialized-value /home/shubham/msan/msan_test.c:8:7 in main
Exiting
```

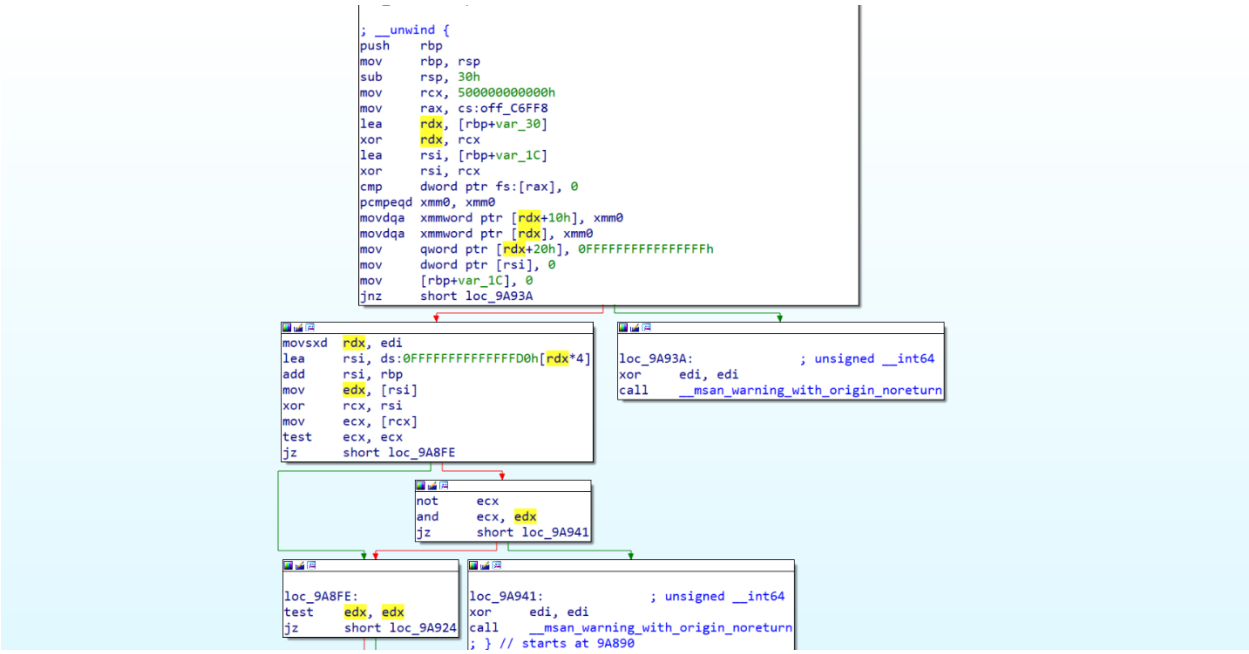
You will observe that MSAN has identified the utilization of an uninitialized value in the line that contains *if(a[argc])*. MemorySanitizer is capable of retracing each uninitialized value back to the memory allocation where it originated from, and utilize this data in the generated reports. This functionality is activated by using the *-fsanitize-memory-track-origins* flag.

Let's see how the program differ in IDA:

Before:



After



Limitations of MSAN

MSAN does it's job pretty well without any False negatives or false positives, only drawback it holds similar to other such tool is it's performance overhead which limit it to use only in testing environment. It takes 2x more memory as it needs 1:1 shadow memory mapping and the execution time get increased to 2.5x due to instrumentation and runtime library work.

Other drawback of MSAN is its limited support to clang and linux environments.

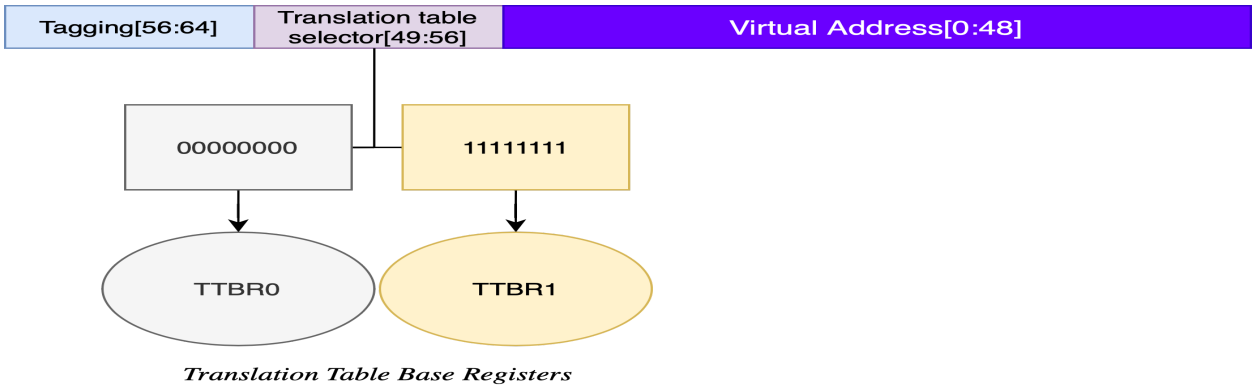
Resources:
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43308.pdf

MTE (Memory tagging Extension)

In 2018, ARM unveiled MTE in ARMv8.5-A, a hardware-enforced memory violation feature for the ARM architecture. It is referred to as a replacement for sanitizers due to its close resemblance in functionality, but with significantly reduced overhead. Unlike all the other tools mentioned in this chapter, MTE is currently the only tool being utilized in production(reference).

MTE internals

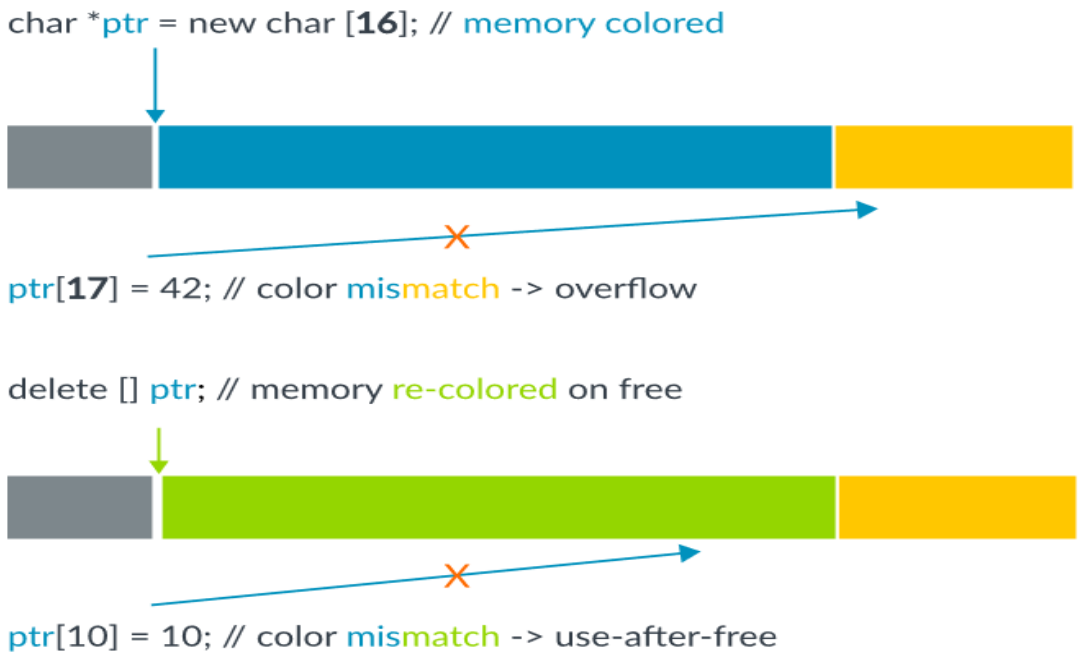
MTE is based on the concept of tagging. To store tags in virtual addresses, It uses the "Top byte ignore" bits in 64 bit *AARCH64* addressing.



Translation Table Base Registers

At high level, when MTE is turned on, memory locations are tagged by adding four bits of metadata to each 16 bytes of physical memory. On memory store/allocation, the same tag is moved into the virtual

address. During dereference/loading, tag are matched if they are same or not. The whole concept can be understood better using the illustration below.



To use MTE in linux(currently only supported by llvm), compile and link your program with *-fsanitize=memtag* flag. This will only work when targeting AArch64 with MemTag extension. One possible way to achieve that is to add *-target aarch64-linux -march=armv8+memtag* to compilation flags. Below are the common instruction compiler can use to add tags to program.

Instruction	Name	Format
ADDG	Add with Tag	ADDG <Xd/SP>, <Xn/SP>, #<uimm6>, #<uimm4>
CMPP	Compare with Tag	CMPP <Xn/SP>, <Xm/SP>
GMI	Tag Mask Insert	GMI <Xd>, <Xn/SP>, <Xm>
IRG	Insert Random Tag	IRG <Xd/SP>, <Xn/SP>{, <Xm>}
LDG	Load Allocation Tag	LDG <Xt>, [<Xn/SP>{, #<simm>}]
LDGV	Load Tag Vector	LDGV <Xt>, [<Xn/SP>]!

Resources:

https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

https://8ksec.io/arm64-reversing-and-exploitation-part-10-intro-to-arm-memory-tagging-extension-mte/

Future of Memory corruptions and it's mitigations

Examining the evolution of memory corruption vulnerabilities reveals a notable shift in recent years. The collective awareness within the security industry and among developers has significantly increased, leading to a decline in the prevalence of straightforward memory corruptions. However, these vulnerabilities persist in more intricate forms. Notably, the realms of Linux and Windows kernelspace continue to harbor traditional memory corruption vulnerabilities, albeit in decreasing numbers.

Despite the persistence of these vulnerabilities, the implementation of robust mitigations has made exploiting memory corruptions progressively challenging. As security measures become more sophisticated, the traditional avenues for attacks are closing, pushing adversaries towards exploiting complex and less common scenarios.

In the landscape of mitigations, we currently deploy first-generation defenses that provide protection against a wide range of cases. Second-generation mitigation support is now integrated into most compilers and architectures, although it is not always enabled by default. There remains room for improvement in terms of performance, and efforts are underway to strike the right balance between security and efficiency.

Additionally, the collaboration between hardware and software ecosystems is strengthening, with more emphasis on designing processors and architectures that inherently resist memory corruption exploits. The integration of hardware-enforced security mechanisms, such as Intel's Control-Flow Enforcement Technology (CET) and ARM's Pointer Authentication Codes (PAC), adds an extra layer of defense against memory-based attacks.

Moreover, the adoption of DevSecOps practices is playing a pivotal role in mitigating memory corruption risks. The continuous integration and deployment pipelines incorporate security checks (chapter 3) and automated testing for identifying and addressing vulnerabilities in the early stages of development.

In conclusion, while memory corruption vulnerabilities persist, the landscape is evolving towards a more resilient future. The combination of advanced mitigations, collaborative hardware-software efforts, and proactive development practices positions the cybersecurity community to stay one step ahead in the ongoing cat-and-mouse game with malicious actors.

Memory-safe Programming Languages:

Using memory-safe programming languages, such as Rust or Ada, can provide inherent memory protection by preventing common programming errors like buffer overflows, use-after-free, and null pointer dereferences. These languages incorporate memory safety features into their design and mitigate many memory-related vulnerabilities. Major platform vendors like Microsoft and linux kernel are spending their resources and money to shift the core of their operating system from C/C++ to more robust language rust. It will be interesting to see the outcome in upcoming years.

Mitigation matrix:

The matrix provided below will prove beneficial for both security researchers and developers. Security professionals can utilize it to identify the current mitigations available and the potential challenges they may encounter in real-world applications. Developers can also make use of this matrix to determine the necessary measures to incorporate into their applications while ensuring minimal impact on performance.

Mitigation	Year	Tool/Technique	Targeted vulnerabilities	Hardware assisted	OS support	Default present	Kernel support	Active/Deprecated	Perf impact	Significant Bypasses
BoundCheck	1992	Tool	Stack overflow, Use after free, Double free	No	Windows	No	No	Deprecated	High	No
PageHeap	~1995	Tool	Heap Overflow, Double free, Use after free	No	Windows	No	Yes	Active	High	No
StackGuard	1997	Technique	Stack overflow	No	Linux/Windows	Yes	Yes	Active	Low	Yes
Libsafe/Libverify	2000	Tool	Stack overflow	No	Linux	No	Yes	Deprecated	Low	No
Stack Shield	2000	Tool	Stack overflow	No	Linux	No	No	Deprecated	Not available	No
Stack Ghost	2001	Technique	Stack overflow	Yes	Linux	No	Yes	Deprecated	Low	Yes

Memcheck	2003	Tool	Use after free, Buffer overflow, Illegal read/write, Double free, Memory leaks	No	Linux	No	No	Active	Upto 40%	No
Propolice	2004	Technique	Stack overflow	No	Linux	Yes	No	Deprecated	Low	No
NX Stack	2004	Technique	Stack Overflow	Yes	Linux/Windows	Yes	Yes	Active	Low	Yes
CCFIR/bin-CFI	2005	Tool	Exploitation	No	Linux	No	No	Deprecated	upto 50%	
ASLR	2005	Technique	Exploitation	No	Linux/Windows	Yes	Yes	Active	Low	Yes
Taint Trace	2006	Tool	Stack overflow, Format string, Indirect calls modification	No	Linux	No	No	Deprecated	5x	No
ASAN	2012	Tool	Buffer overflow, Use after free, Null pointer dereferance, Use after return, Uninitialized memory, Memory leaks	No	Linux/Windows	Yes	Yes	Active	Processing upto 73%, Memory usage 230%	No
UBSAN	2013	Tool	OOB read/write, Null pointer dereferance, Integer underflow	No	Linux/Windows	Yes	Yes	Active	Processing upto 3x, Disk usage upto 20x	No
MSAN	2015	Tool	Uninitialized memory	No	Linux	Yes	Yes	Active	2.5x	No
LLVM-CFI	2014	Technique	Exploitation	No	Linux	Yes	Yes	Active	VTV- upto 20% IFCC - upto 4%	Yes
CFG	2014	Technique	Exploitation	No	Windows	Yes	Yes	Active	Medium-High	Yes
SafeStack	2014	Technique	Exploitation	No	Linux	No	No	Active	Low - max untime overhead	Yes

									3.0%, memory overhead 5.3%	
ACG	2016	Technique	Exploitation	Yes	Windows	Yes	Yes	Active	Low	Yes
PAC	2018	Technique	Exploitation	Yes	Linux/Windows	Yes	Yes	Active	Low	Yes
BTI	2018	Technique	Exploitation	Yes	Linux/Windows	Yes	Yes	Active	Low	Yes
MTE	2019	Technique	Buffer overflow, Heap overflow, Use after free, Double free, Null pointer dereferance	Yes	Linux/Windows	Yes	Yes	Active	Low-Medium	No
XFG	2019	Technique	Exploitation	No	Windows	No	No	Active	Low-Medium	No
IBT	2020	Technique	Exploitation	Yes	Linux/Windows	Yes	Yes	Active	Low	Yes
Shadow stack	2020	Technique	Exploitation, Buffer overflow	Yes	Linux/Windows	Yes	Yes	Active	Low	Yes
FGKASLR	2020	Technique	Exploitation	No	Linux	Yes	Yes	Active	Low	No
FineIBT	2021	Technique	Exploitation	Yes	Linux	Yes	Yes	Active	Low	No
KCFI	2022	Technique	Exploitation	No	Linux	Yes	Yes	Active	Low	No

	first generation mitigations
	second generation mitigations
	Error detection tools