presented by

UEFI Security Defenses

UEFI Summer Summit – July 16-20, 2012

Presented by Dick Wilkins

Dick_Wilkins@phoenix.com

Phoenix Technologies Ltd.

Agenda

- Introduction
- Defining the Problem
- Defensive Security Goals
- Stack Buffer Overrun Detection (/GS, /RTC)
- Heap Corruption Detection
- Data Execution Prevention (DEP) / No eXecute (NX)
- Address Space Location Randomization (ASLR)
- Conclusion
- Q & A

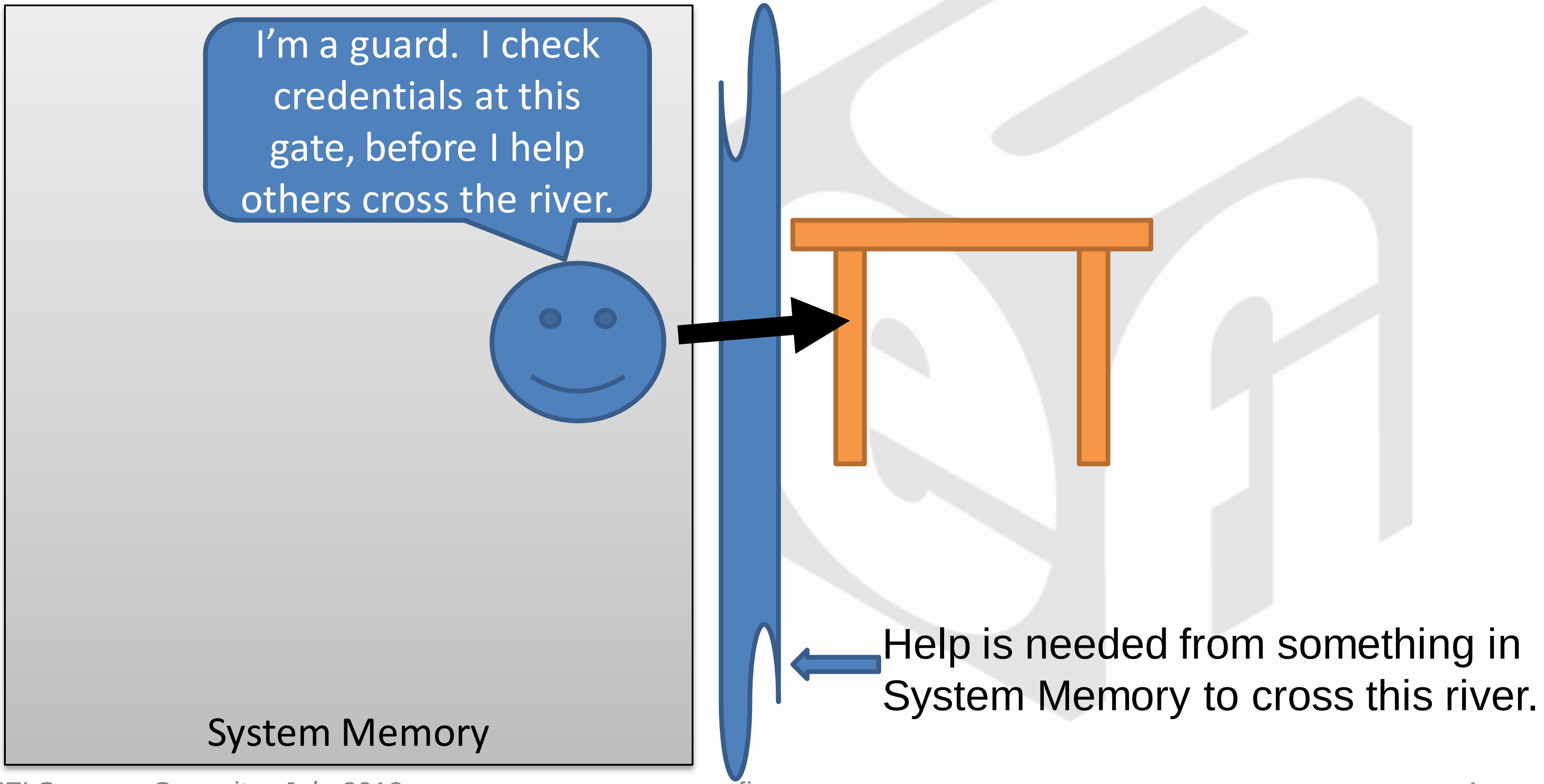
Introduction

We will be discussing security defenses that harden UEFI BIOS implementations against attacks

The defenses discussed here have been added to EDK 2 as part of a collaboration between Microsoft and Phoenix Technologies Ltd.

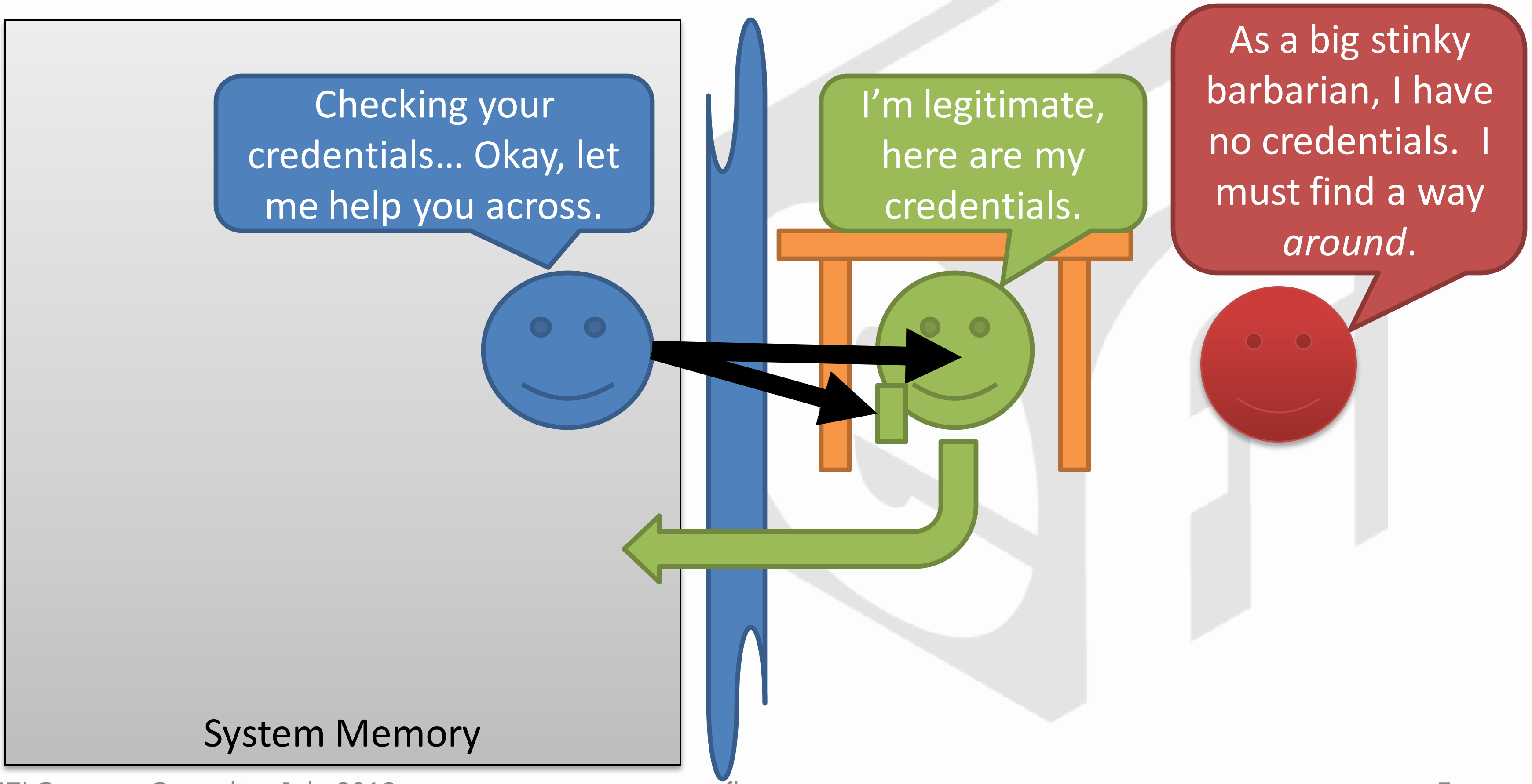
Defining the Problem

- Imagine the BIOS as a guarded gateway


I'm a guard. I check credentials at this gate, before I help others cross the river.
System Memory
Help is needed from something in System Memory to cross this river.

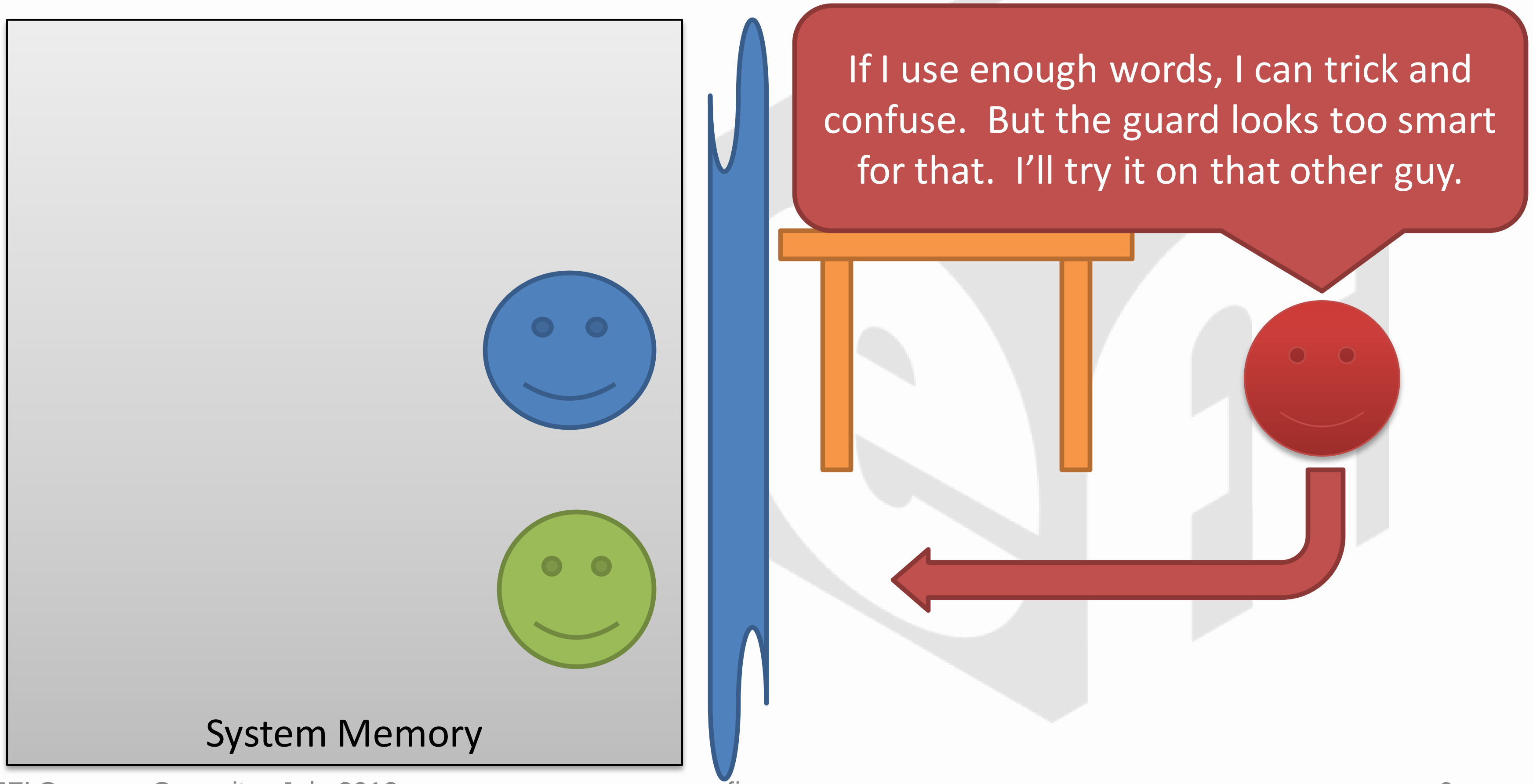
Defining the Problem

- Guards are good at checking credentials


Checking your credentials… Okay, let me help you across.
I'm legitimate, here are my credentials.
As a big stinky barbarian, I have no credentials. I must find a way *around*.
System Memory

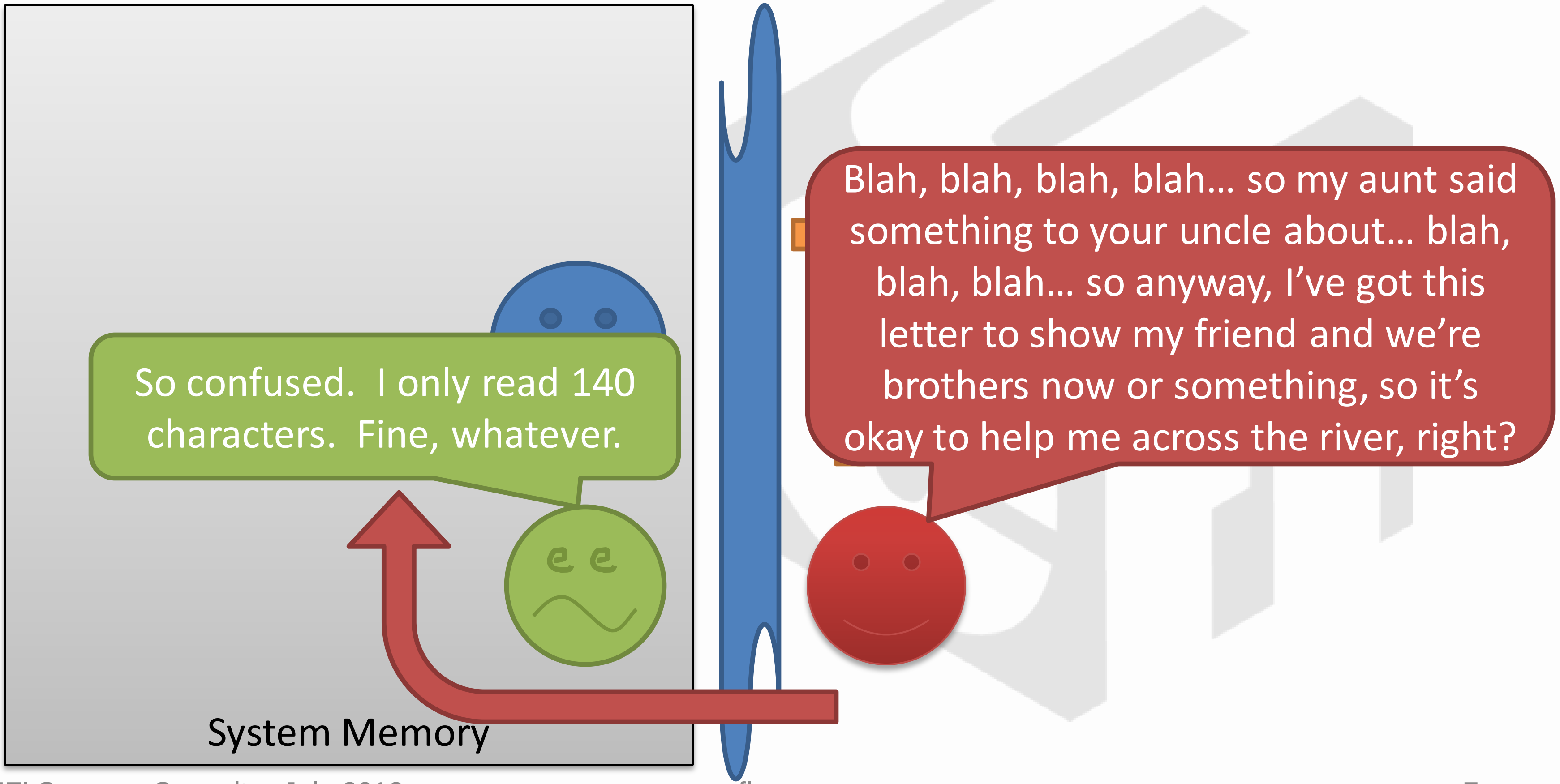
Defining the Problem

- Attackers can't get through the gate


If I use enough words, I can trick and confuse. But the guard looks too smart for that. I'll try it on that other guy.
System Memory

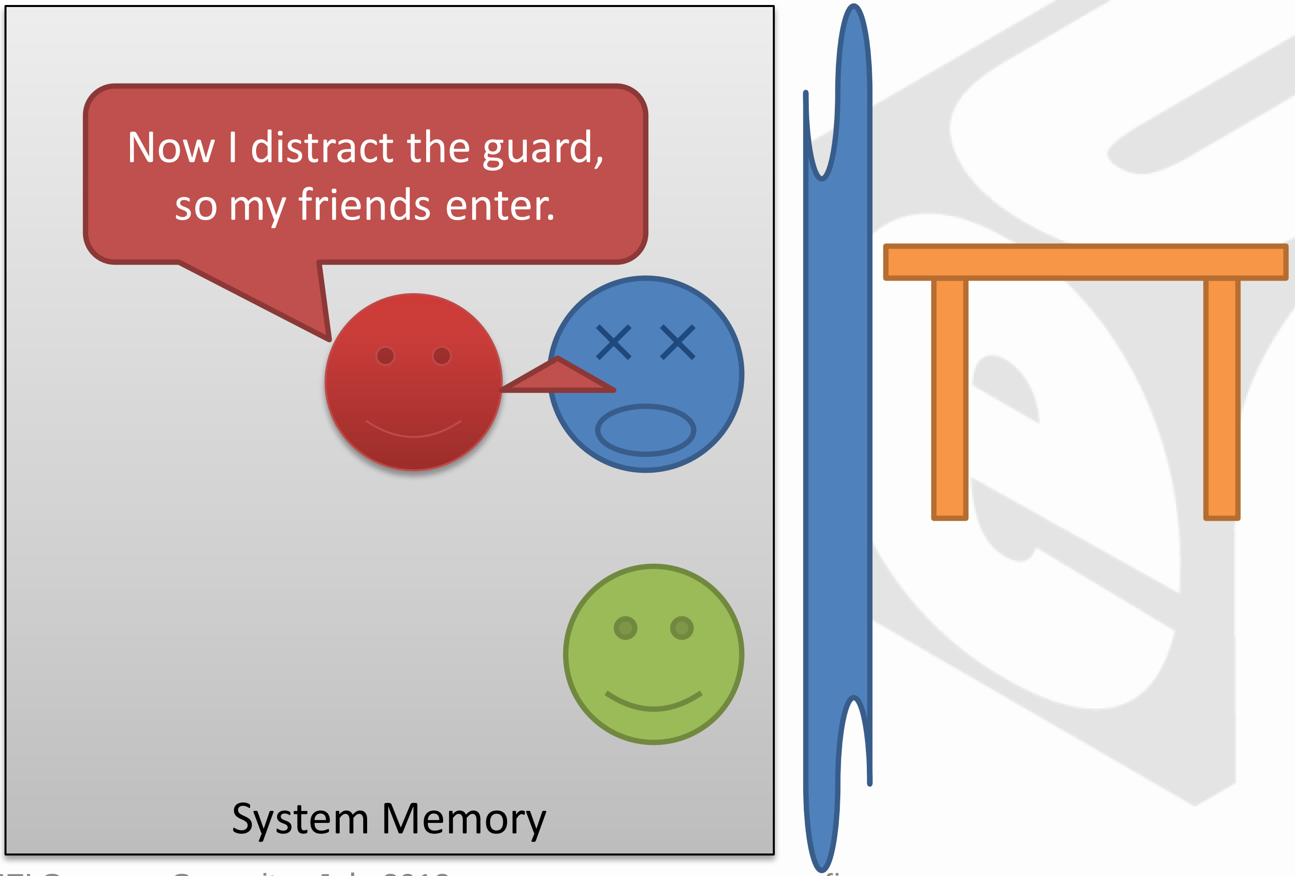
Defining the Problem

- If it isn't a guard, what does it check?


So confused. I only read 140 characters. Fine, whatever.
Blah, blah, blah, blah… so my aunt said something to your uncle about… blah, blah, blah… so anyway, I've got this letter to show my friend and we're brothers now or something, so it's okay to help me across the river, right?
System Memory

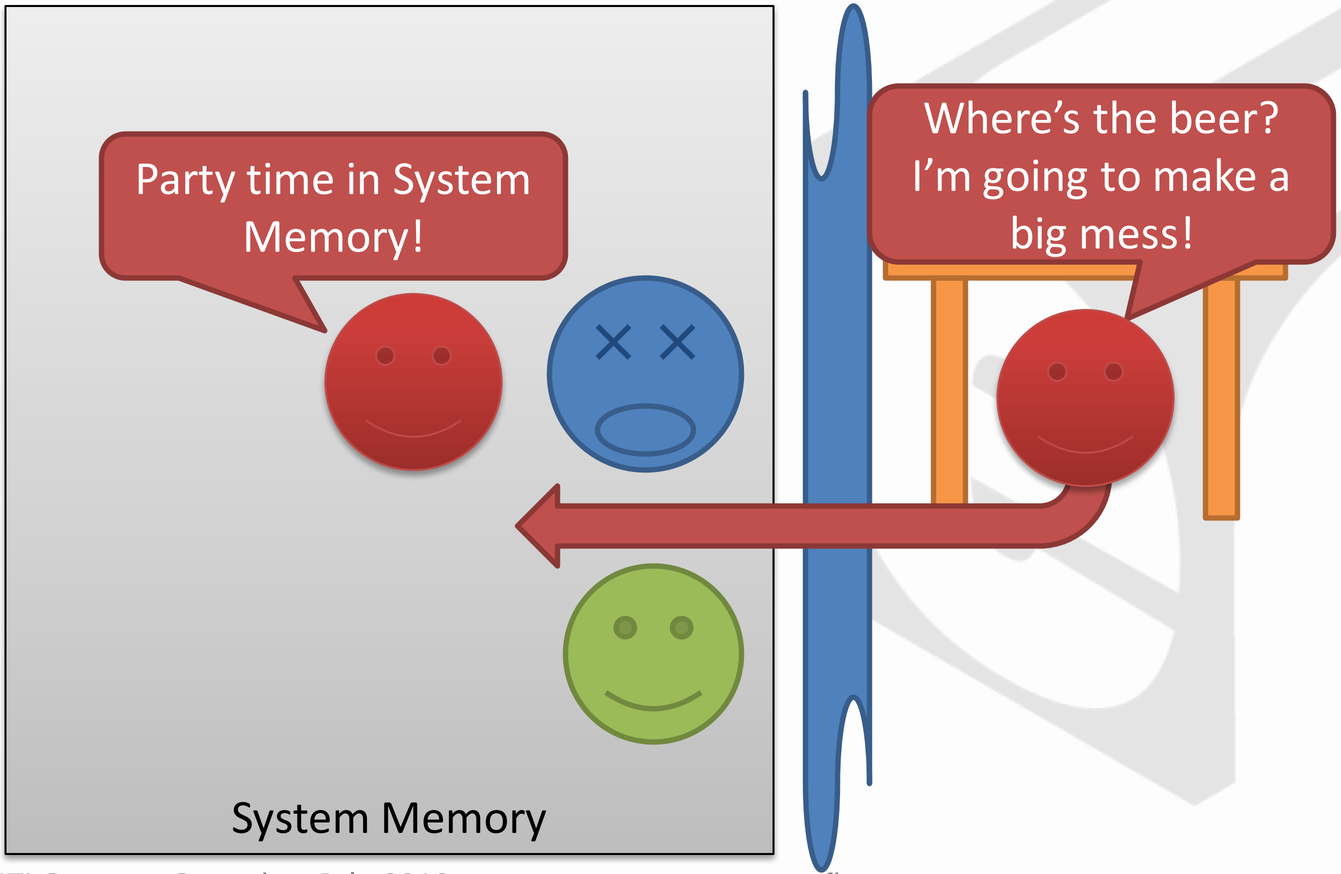
Defining the Problem

- If the attacker distracts the guard…



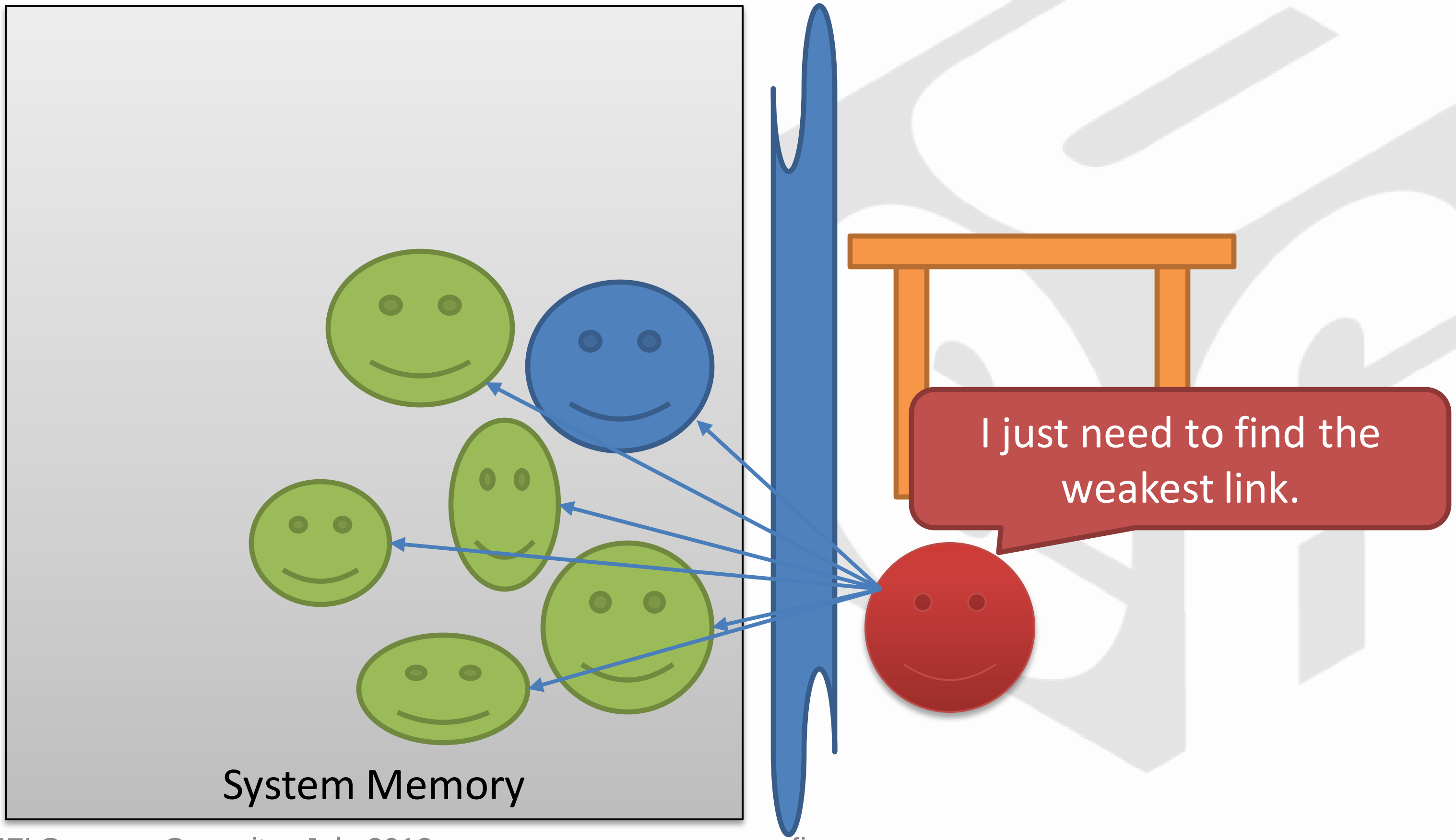
Defining the Problem

- We have to prevent this from happening!


Party time in System Memory!
Where's the beer? I'm going to make a big mess!
System Memory

Defining the Problem

- Too much code to be sure it's all perfect



Defining the Problem

- In programming, weak links are generally due to buffer overruns that allow attackers to execute arbitrary code
  - Buffers on the stack are adjacent to the function return address and local variables
  - Buffers in the heap are adjacent to protocol definitions and executable code

Defensive Security Goals

- The goal is to mitigate the effects of buffer overruns in the following ways
  - Detect Buffer Overruns corrupting the stack with /GS and /RTCs compiler switch support
  - Detect Buffer Overruns corrupting the Heap by verifying heap pool object signatures
  - Prevent usage of data buffers as storage for exploit code by preventing execution of data
  - Prevent attackers from exploiting valid code loaded at a known address through address space location randomization (ASLR)

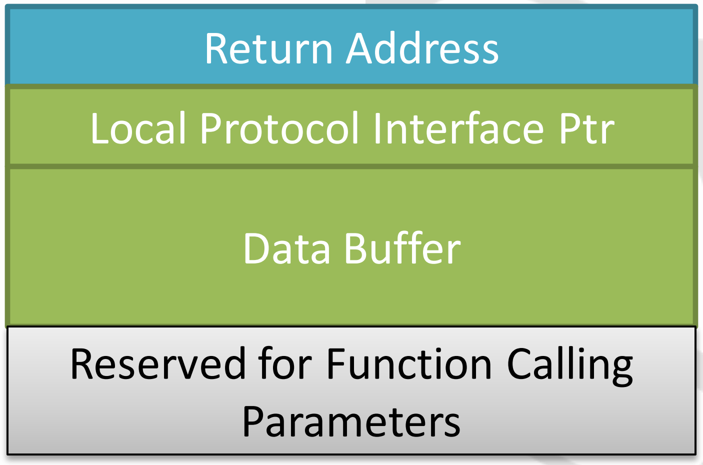
Protecting from Stack Buffer Overruns

Stack Buffer Overrun Detection

- Goal: Detect Buffer Overruns Corrupting the Stack
  - Local Variables are stored on the stack
  - Function return addresses are stored on the stack
- The intent of buffer overrun detection is to expose coding errors at runtime that could compromise security, not to provide complete protection from all possible buffer overrun attacks

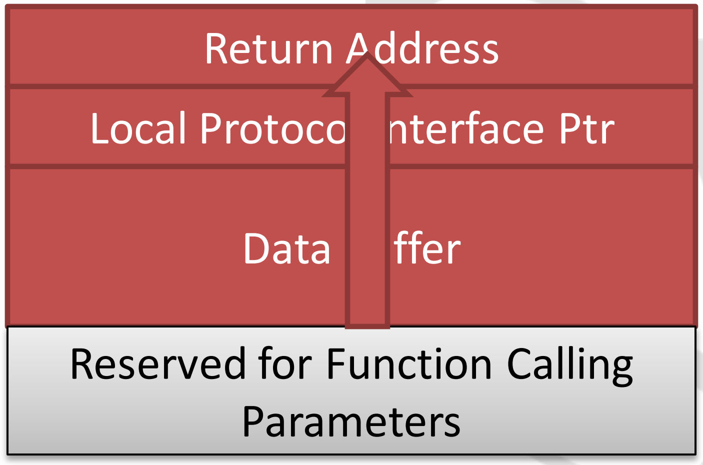
Stack Buffer Overrun Detection

- Illustration of a vulnerable stack frame


Return Address
Local Protocol Interface Ptr
Data Buffer
Reserved for Function Calling Parameters

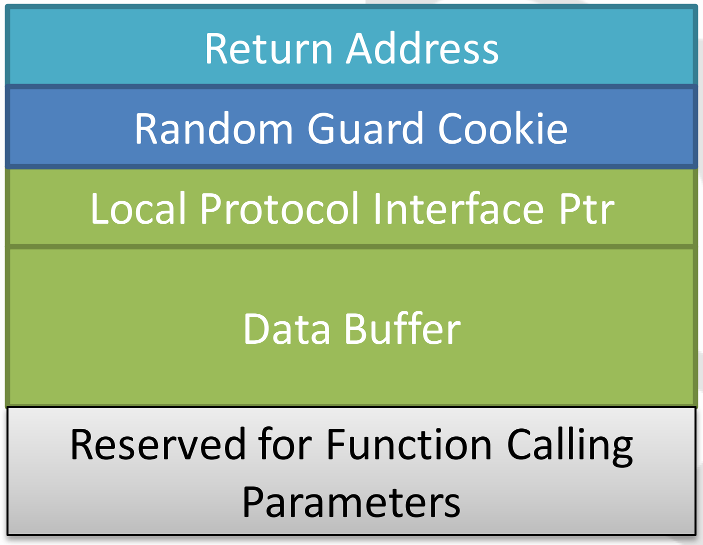
Stack Buffer Overrun Detection

- Buffer overflows occur when a function does not correctly check the amount of data being transferred into a buffer


Return Address
Local Protocol Interface Ptr
Data Buffer
Reserved for Function Calling Parameters

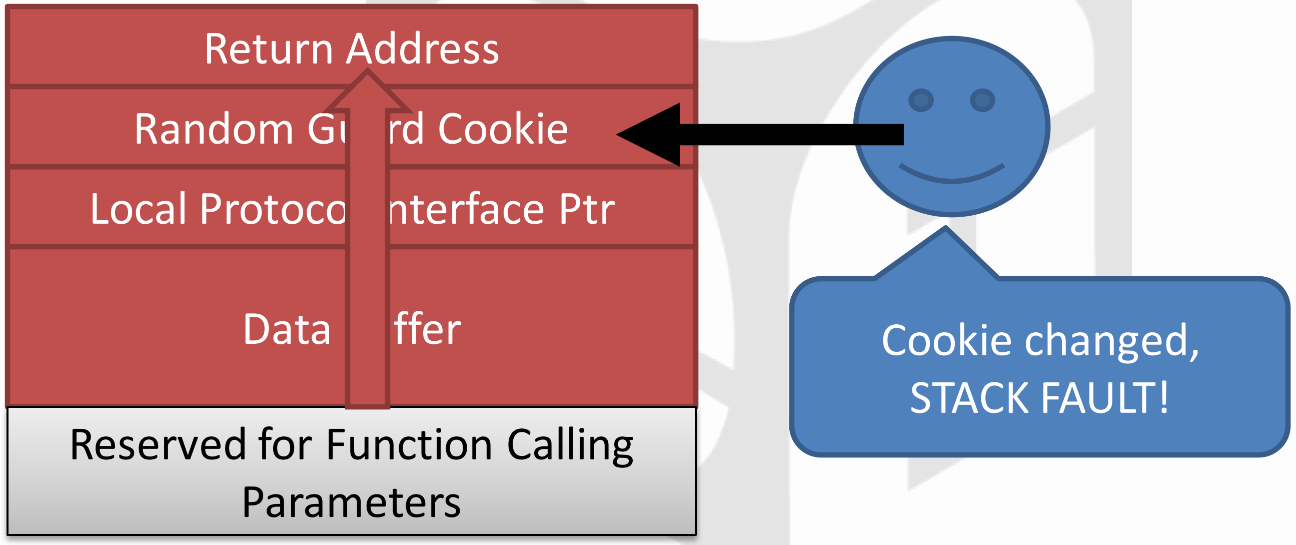
Stack Buffer Overrun Detection

- The MSVC /GS compiler switch inserts a randomized guard cookie onto the stack between the return address and locals


Return Address
Random Guard Cookie
Local Protocol Interface Ptr
Data Buffer
Reserved for Function Calling Parameters

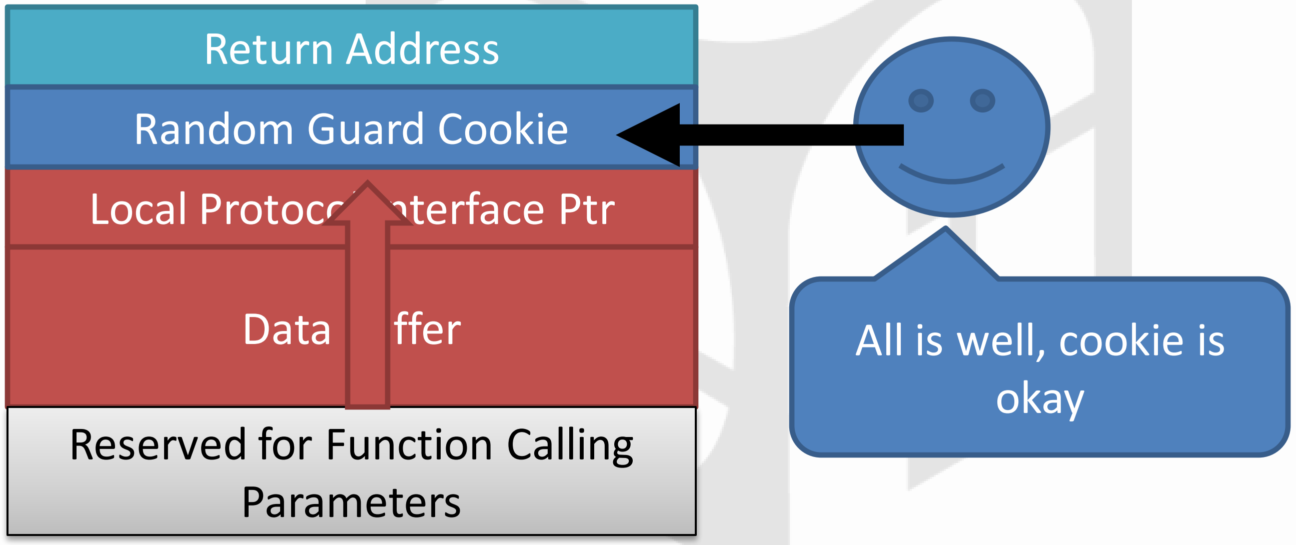
Stack Buffer Overrun Detection

- Changing the return address with a buffer overflow requires changing the guard cookie, so such overflows are detected


Return Address
Random Gu rd Cookie
Local Protoco nterface Ptr
Data ffer
Reserved for Function Calling Parameters
Cookie changed, STACK FAULT!

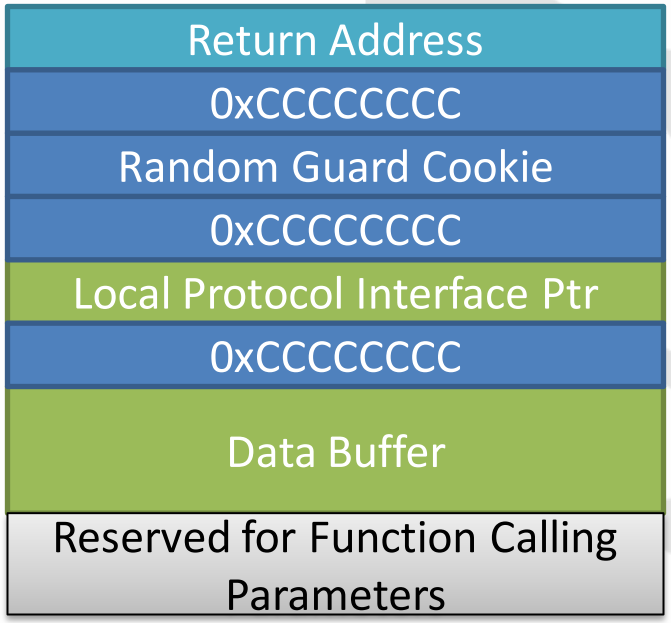
Stack Buffer Overrun Detection

- /GS does NOT detect changes to locals if the buffer overrun doesn't reach the guard cookie


Return Address
Random Guard Cookie
Local Protoc nterface Ptr
Data ffer
Reserved for Function Calling Parameters
All is well, cookie is okay

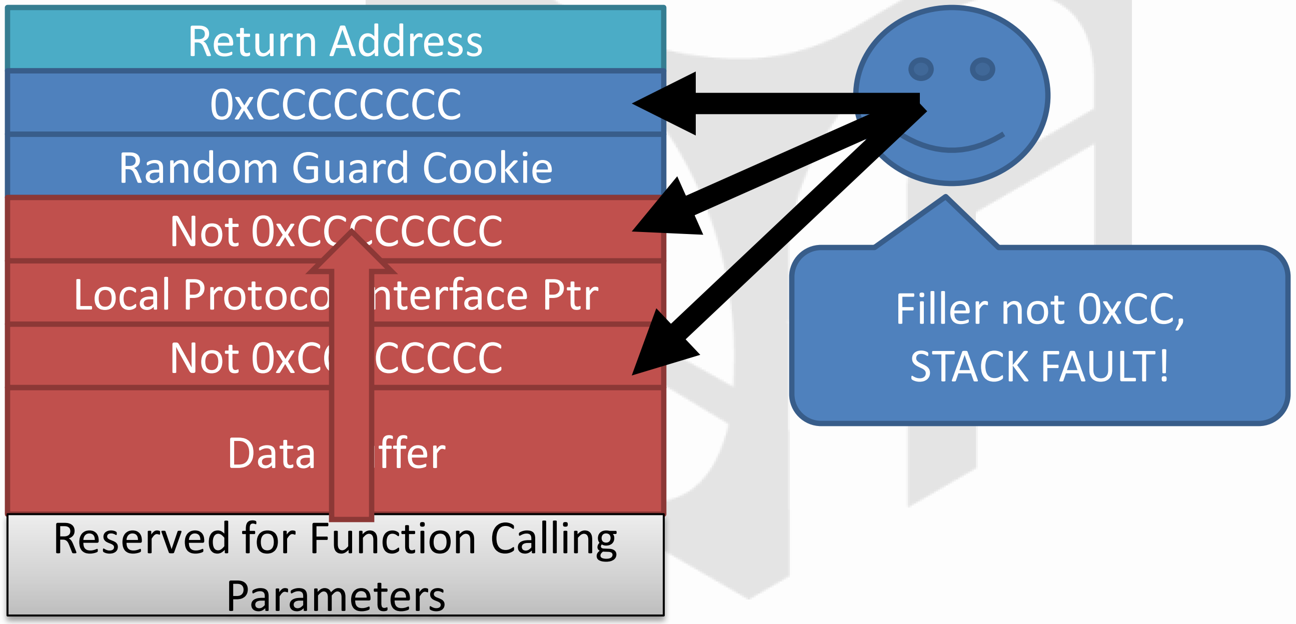
Stack Buffer Overrun Detection

- The MSVC /RTCs compiler switch inserts 0xCC onto the stack between local variables



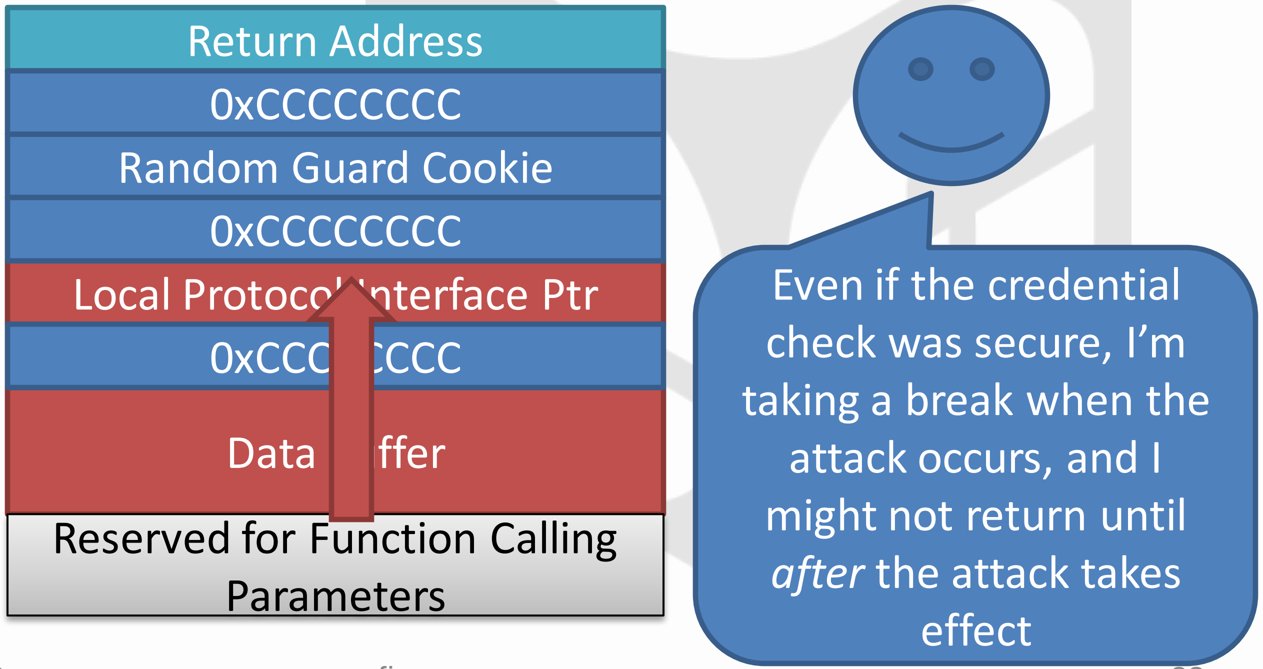
Stack Buffer Overrun Detection

- If a buffer overflow changes the filler between locals, that overflow is detected on return


Return Address
0xCCCCCCCC
Random Guard Cookie
Not 0xCCCCCCCC
Local Protoco nterface Ptr
Not 0xCC CCCCC
Data ffer
Reserved for Function Calling Parameters
Filler not 0xCC, STACK FAULT!

Stack Buffer Overrun Detection

- However, 0xCC is easy to forge and the check comes just before return, not immediately after the buffer overrun


Return Address
0xCCCCCCCC
Random Guard Cookie
0xCCCCCCCC
Local Protocol Interface Ptr
0xCCCCCCCC
Data Buffer
Reserved for Function Calling Parameters
Even if the credential check was secure, I'm taking a break when the attack occurs, and I might not return until *after* the attack takes effect

Stack Buffer Overrun Detection

- Note that /RTC switches require that optimizations be disabled
- Because of the constant signature and the optimization disable requirement, the current /RTC implementation should be considered a debugging feature meant to help identify buffer overflows, and does not provide much more security in a release build than is already provided by /GS

Stack Buffer Overrun Detection

- Phoenix Stack Buffer Overrun Detection Implementation
  - New BaseStackCheckLib MdePkg library
    - Contains support for /GS compiler switch
    - Contains support for /RTCs compiler switch
  - BaseStackCheckLib must be linked to all object code compiled with the /GS or /RTCs switches set

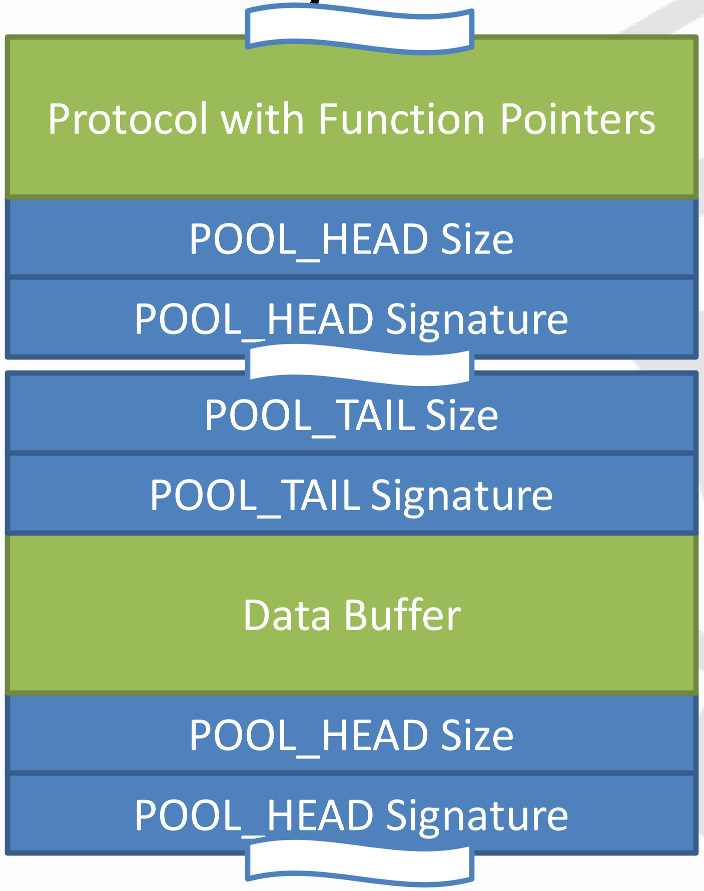
Heap Corruption Detection

Heap corruption detection

- Goal: Detect Buffer Overruns Corrupting the Heap
  - Protocols with function pointers are typically stored on the heap
  - Dynamically sized buffers are also usually allocated and stored on the heap

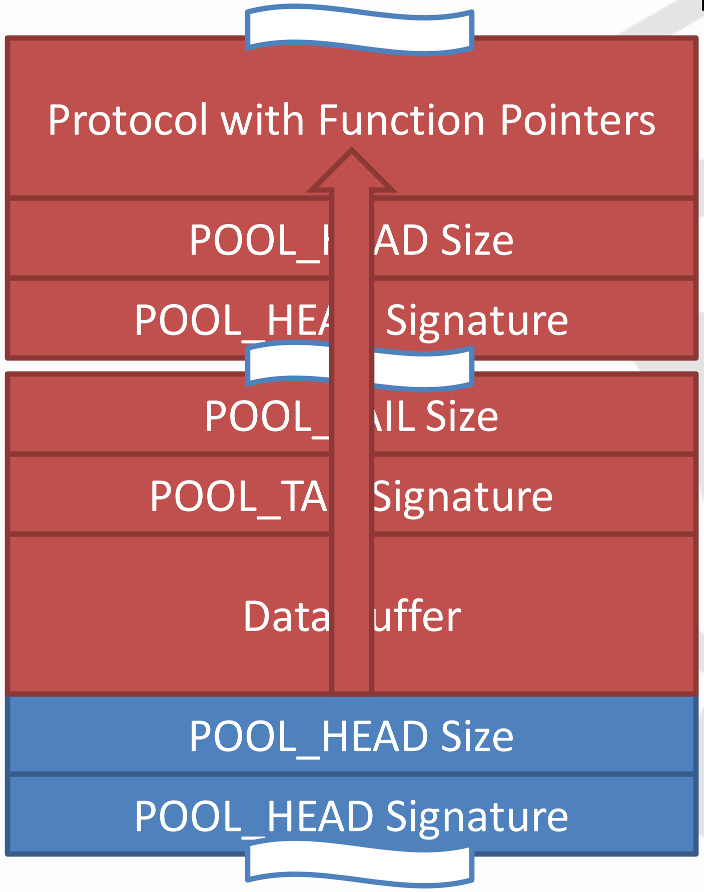
Heap corruption detection

- Heap is dynamically allocated memory



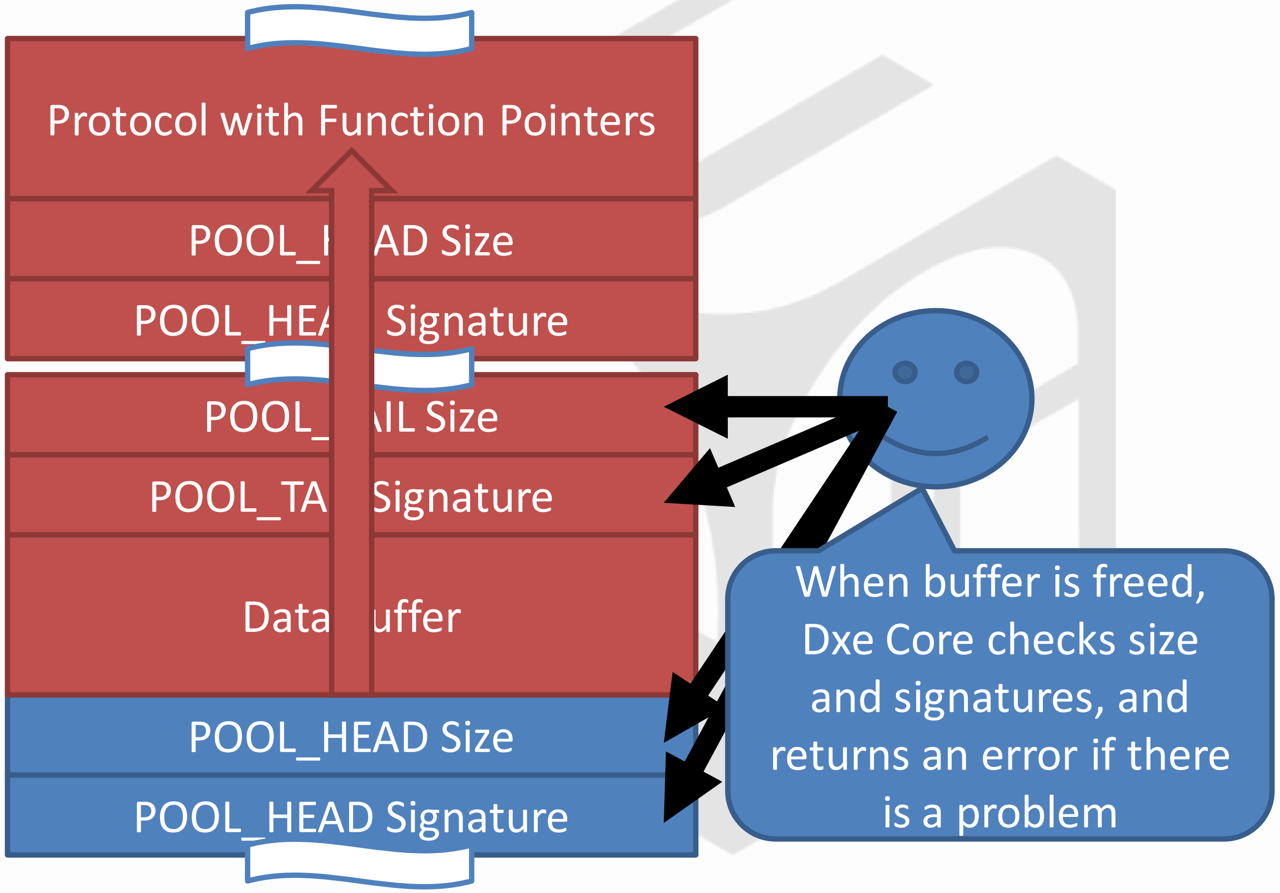
Heap corruption detection

- Buffer overrun attack on Heap looks like



Heap corruption detection

- Existing signature checks should catch


Protocol with Function Pointers
POOL_HEAD Size
POOL_HEAD Signature
POOL_TAIL Size
POOL_TAIL Signature
Data Buffer
POOL_HEAD Size
POOL_HEAD Signature
When buffer is freed, Dxe Core checks size and signatures, and returns an error if there is a problem

Heap corruption detection

- Remaining ways to improve heap corruption detection
  – More of the pool objects in the heap should be verified, outside of just those signatures around specific pool objects checked during heap free or allocate calls
  – Full validation of heap should occur periodically, outside the context of allocate and free calls
  – Heap signatures should be encrypted at run-time using XOR with a random number to prevent signature forgery by attackers
  – Failed signature checks should throw an exception, rather than returning an error, as few clients of “free” function calls check for or handle error conditions returned by free
  – Place Guard Pages (which are pages that are write protected or for which page presence bit is clear) between code and data pages

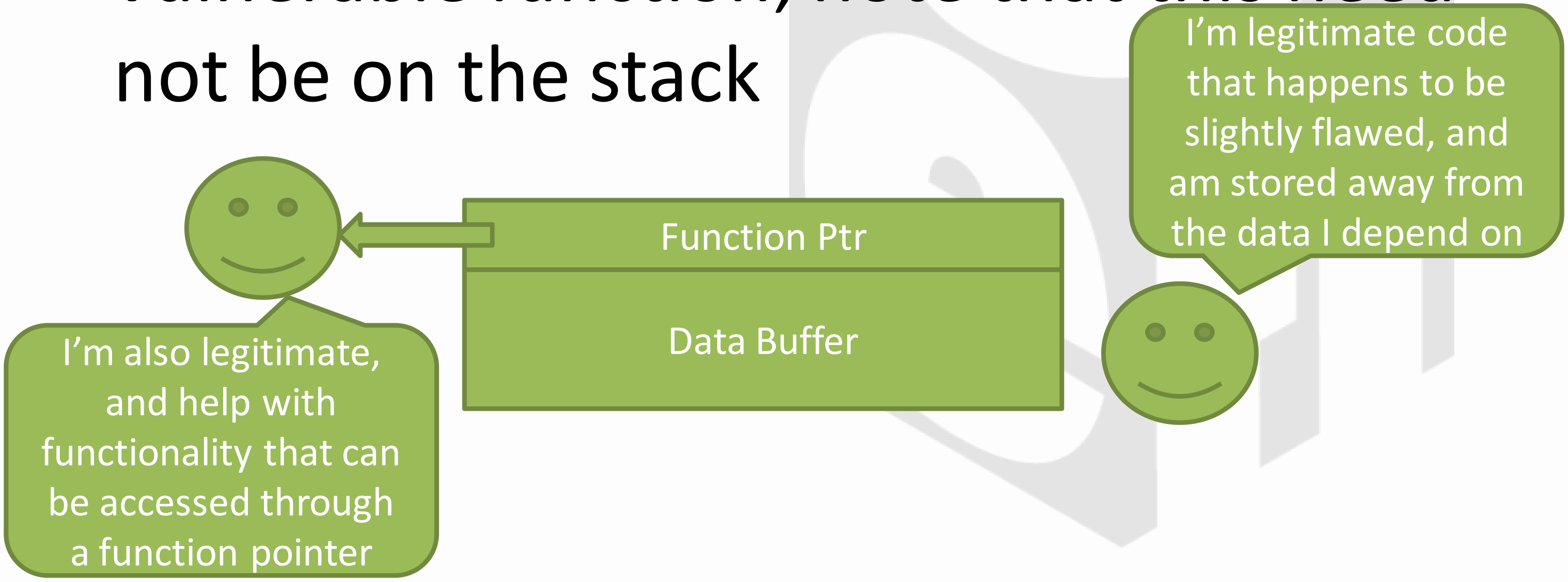
Prevention of the Execution of Data

Data Execution Prevention

- Goal: Prevent usage of data buffers as storage for exploit code
- Enabling CPU Technology
  – Modern x86 CPUs provide support for NX as a bit that can be set in PAE and IA32E page tables (called XD in Intel Volume 3)
  – Setting the execute disable bit in a page table entry causes the processor to page fault when fetching code from the associated page

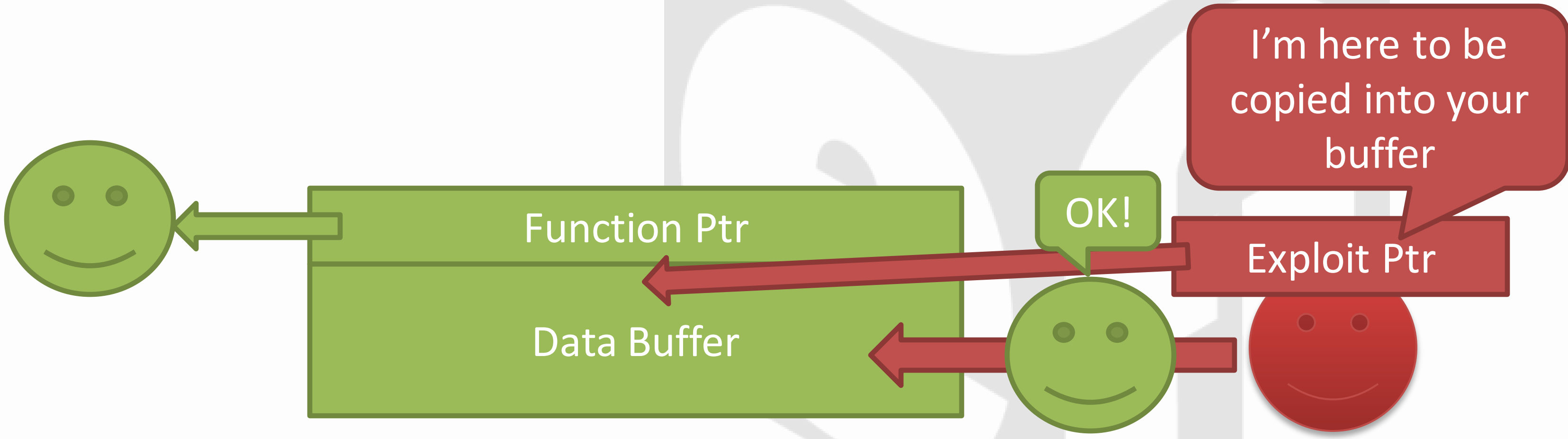
Data Execution Prevention

- Illustration of a vulnerable variable arrangement in the context of a vulnerable function; note that this need not be on the stack


I'm legitimate code that happens to be slightly flawed, and am stored away from the data I depend on
Function Ptr
Data Buffer
I'm also legitimate, and help with functionality that can be accessed through a function pointer

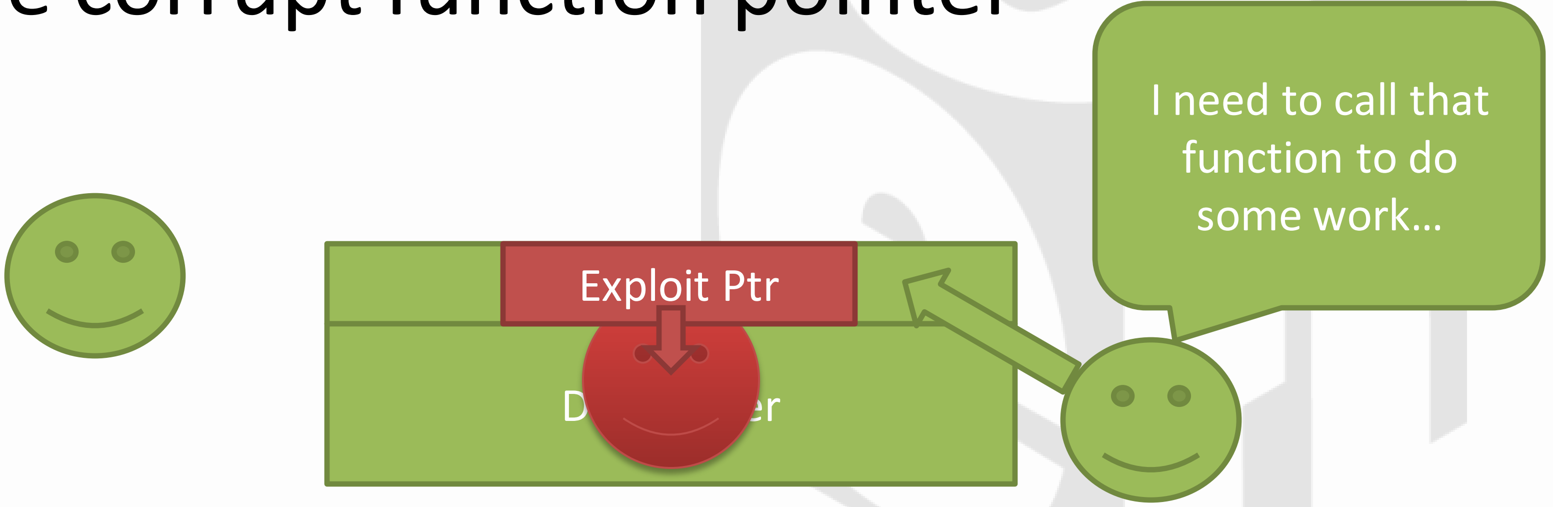
Data Execution Prevention

- The attacker tricks vulnerable code into copying an exploit into the buffer and altering the function pointer to point to it



Data Execution Prevention

- Some function, perhaps the vulnerable function, calls the exploit code through the corrupt function pointer

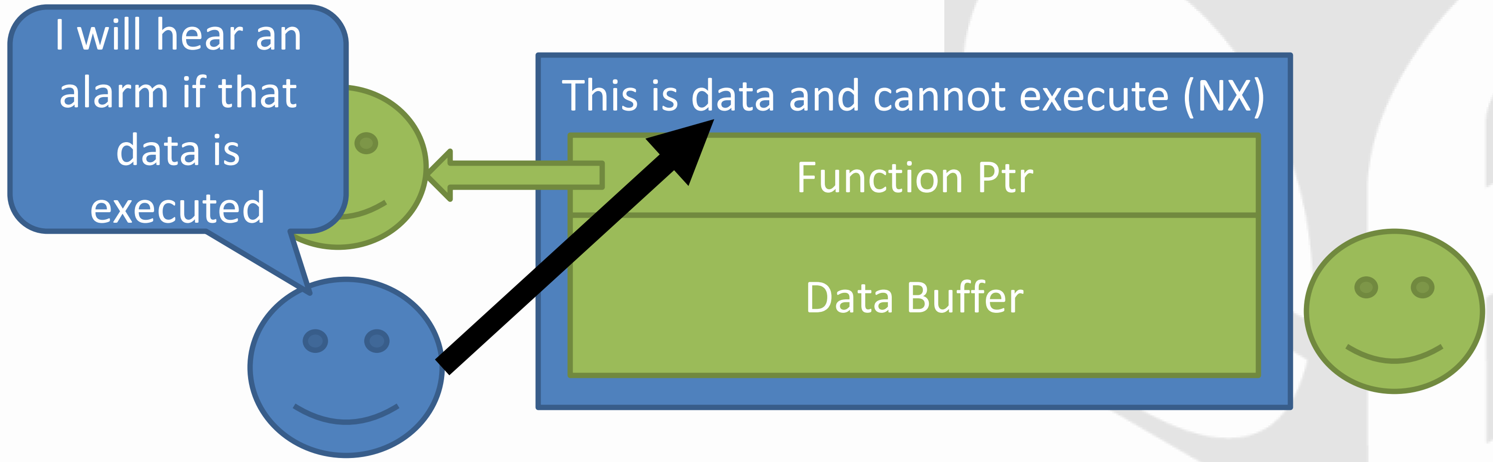


Data Execution Prevention

- The exploit code now has control and can do nearly anything

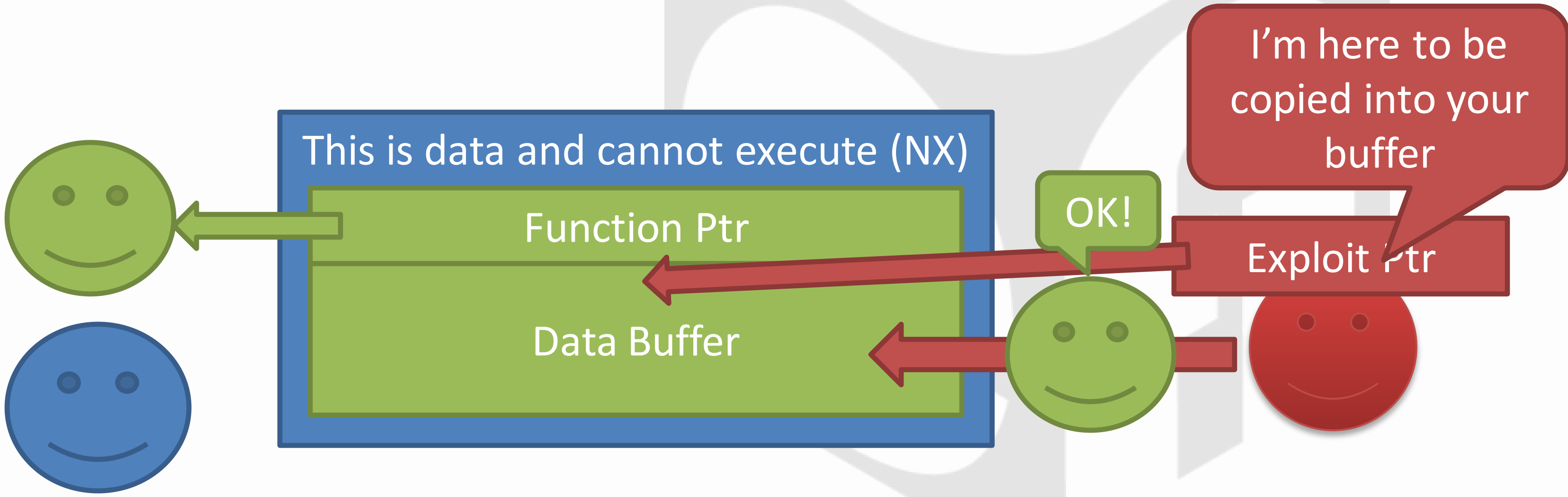
Data Execution Prevention

- With data execution prevention, DXE Core marks all memory that is definitely data as “NX” or No eXecute


I will hear an alarm if that data is executed
This is data and cannot execute (NX)
Function Ptr
Data Buffer

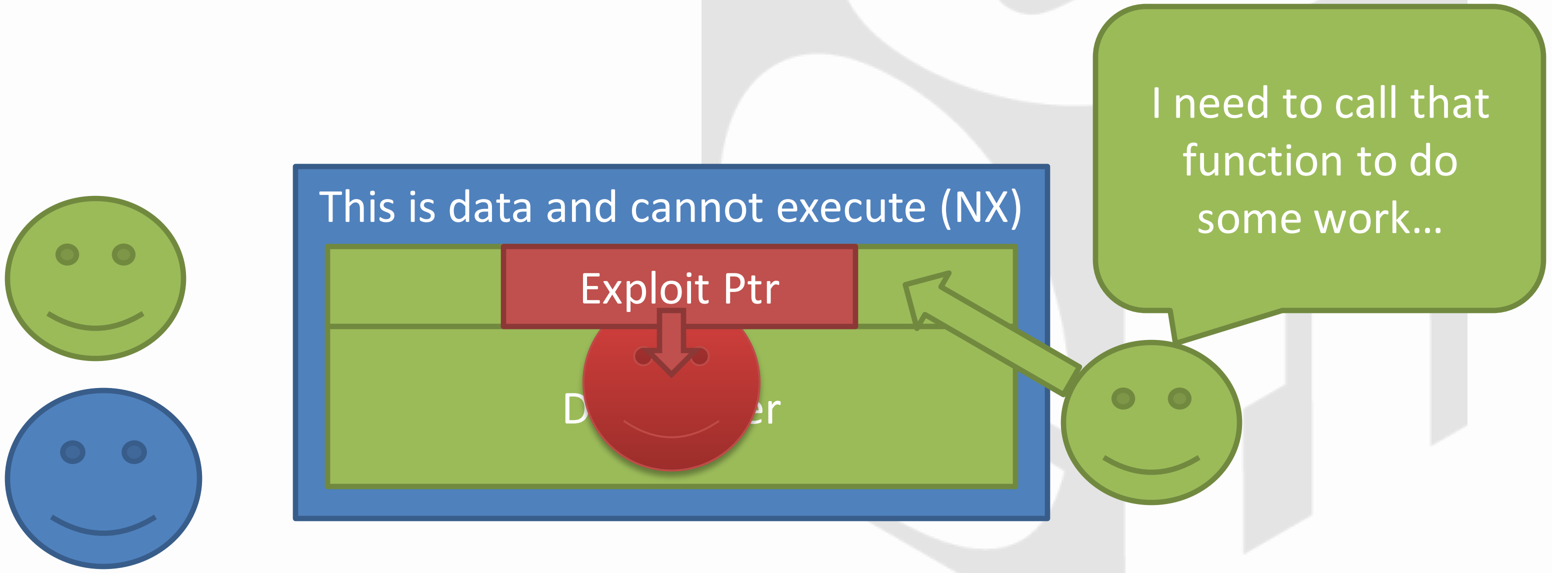
Data Execution Prevention

- The exploit code is still copied into the buffer by the vulnerable code


This is data and cannot execute (NX)
Function Ptr
Data Buffer
OK!
I'm here to be copied into your buffer
Exploit Ptr

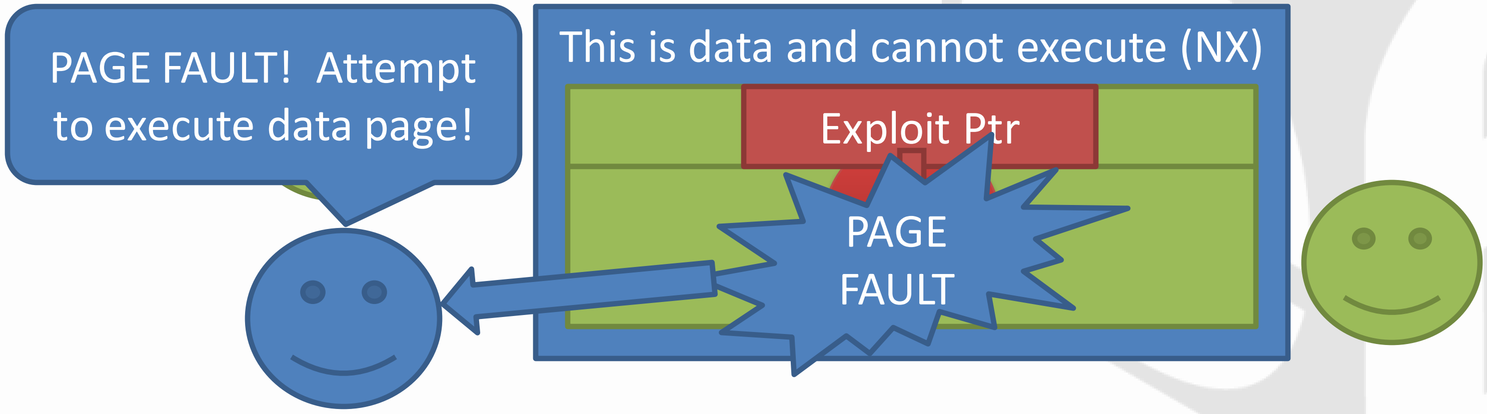
Data Execution Prevention

- Somewhere, the exploit is still called through the modified function pointer


This is data and cannot execute (NX)
Exploit Ptr
D er
I need to call that function to do some work…

Data Execution Prevention

- But the (NX) protection on the data pages acts as an alarm, and the page fault handler in DXE core is called before the exploit can execute


PAGE FAULT! Attempt to execute data page!
This is data and cannot execute (NX)
Exploit Ptr
PAGE FAULT

Data Execution Prevention

- Pre-requisites
  - Memory from which code is to be executed must be allocated as one of the following types
    - EfiReservedMemoryType
    - EfiLoaderCode
    - EfiBootServicesCode
    - EfiRuntimeServicesCode
    - EfiACPIMemoryNVS
  - IA32_EFER.NXE (MSR 0xC0000080 bit 11) must be set for the BSP and all APs when IA32_EFER.LME (MSR 0xC0000080 bit 8) is set

Data Execution Prevention

- Phoenix NX Implementation
  - New BasePageTableLib MdePkg library
    - BasePageTableLib contains stub functions
    - BasePageTableLibIA32E contains IA32E page table support (used for X64)
  - Enabling PcdPageTableNxSupport causes DxeIplPeim to enable IA32_EFER.NXE
  - Enabling PcdPageTableLibrarySupport causes DxeCore to call the Page Table Library functions when pages are allocated

Data Execution Prevention

- Platform and Silicon Considerations
  - All application processor (AP) entry vector setup code that sets bit 8 of MSR 0xC0000080 must also set bit 11 before enabling paging using the boot processors (BP) page tables
  - Very early SMM initialization code re-uses the boot processors page tables
    - AllocatePages must be used to set memory from 0x38000 to 0x40000 to EfiReservedMemoryType during the first SMI
    - The code that sets bit 8 of MSR 0xC0000080 must also set bit 11 when the first SMI occurs
- You will know if you missed anything
  - System will reboot or lock up, depending on current IDT and the conditions of the fault
  - A fetch from data address space will cause a page fault

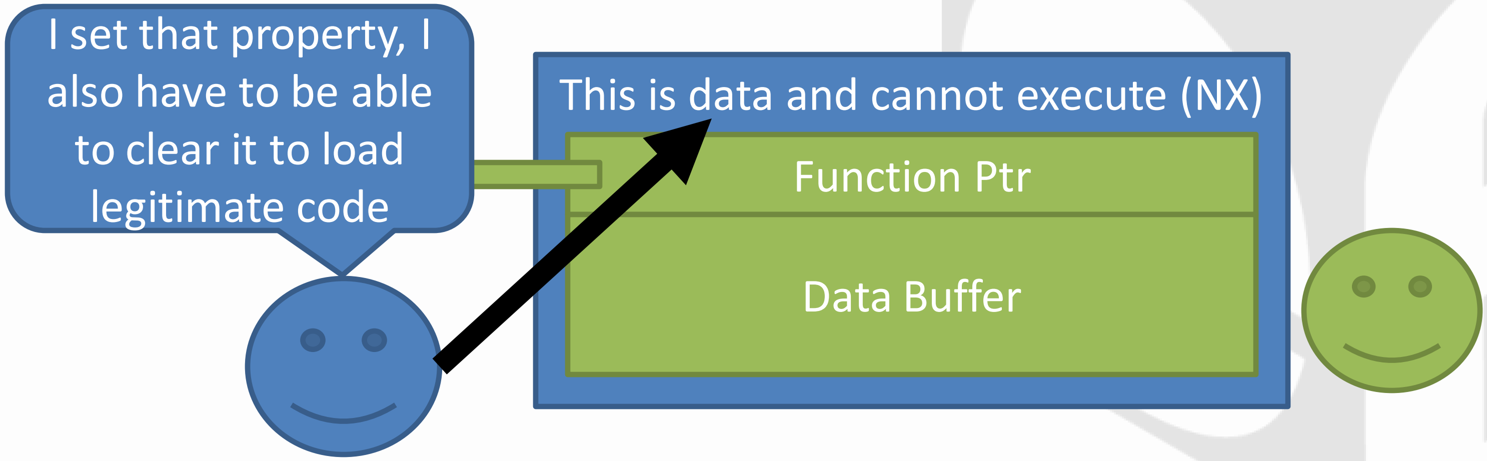
Address Space Layout Randomization (ASLR)

Address Space Layout Randomization (ASLR)

- Goal: Prevent Attacker from Exploiting Valid Code Loaded at a Known Address
- Enabling Technology
  - A good source of random numbers is needed that varies on every boot
  - Random numbers can come from a TPM or the CPU's time stamp counter can be used to seed a random number generator

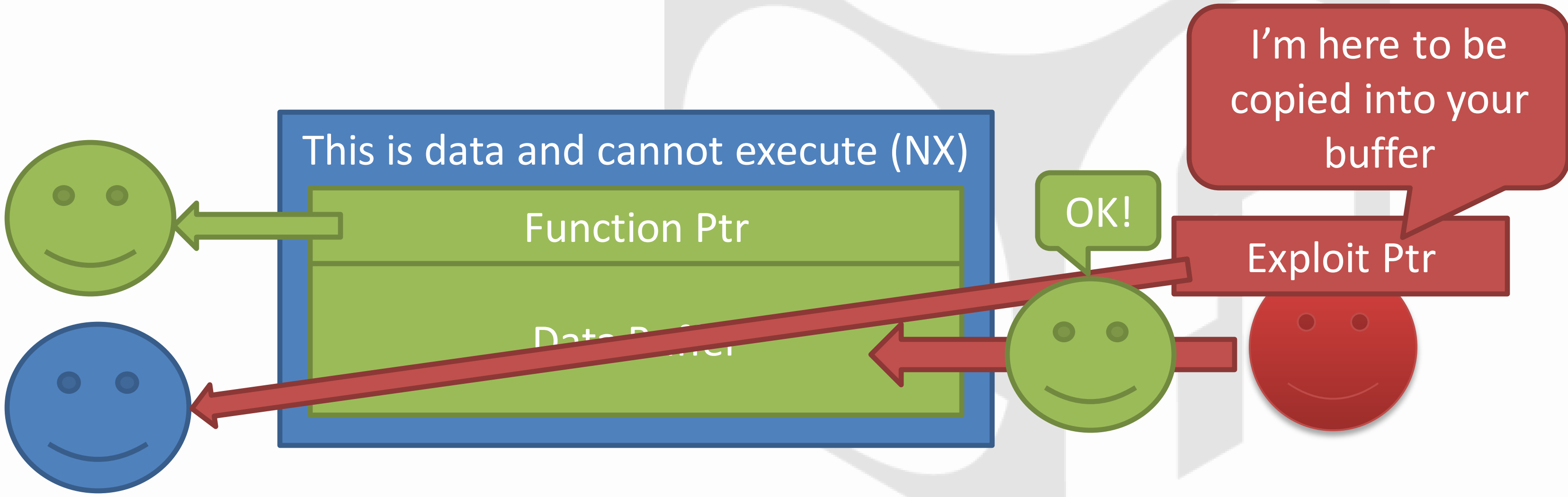
Address Space Layout Randomization (ASLR)

- DXE core contains code to set page table properties, such as NX, as well as to handle page faults


I set that property, I also have to be able to clear it to load legitimate code
This is data and cannot execute (NX)
Function Ptr
Data Buffer

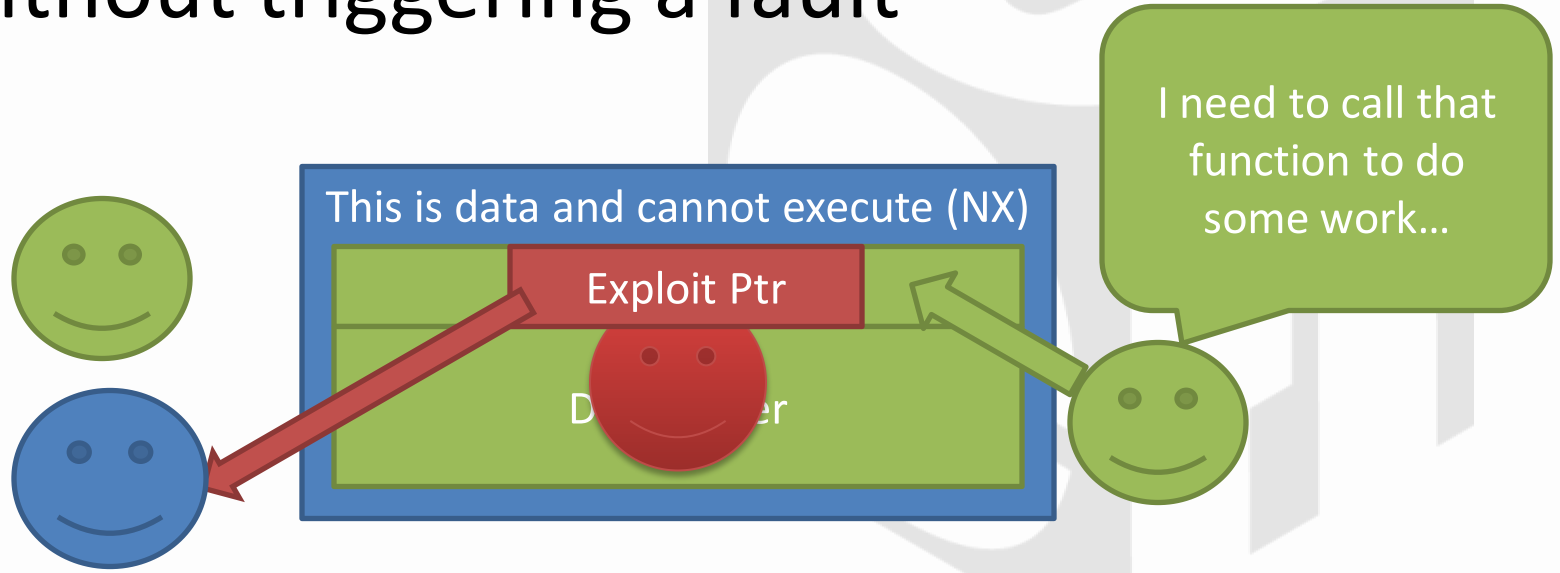
Address Space Layout Randomization (ASLR)

- The attacker once again overflows the buffer, and changes a function pointer


I'm here to be copied into your buffer
This is data and cannot execute (NX)
Function Ptr
Data Buffer
OK!
Exploit Ptr

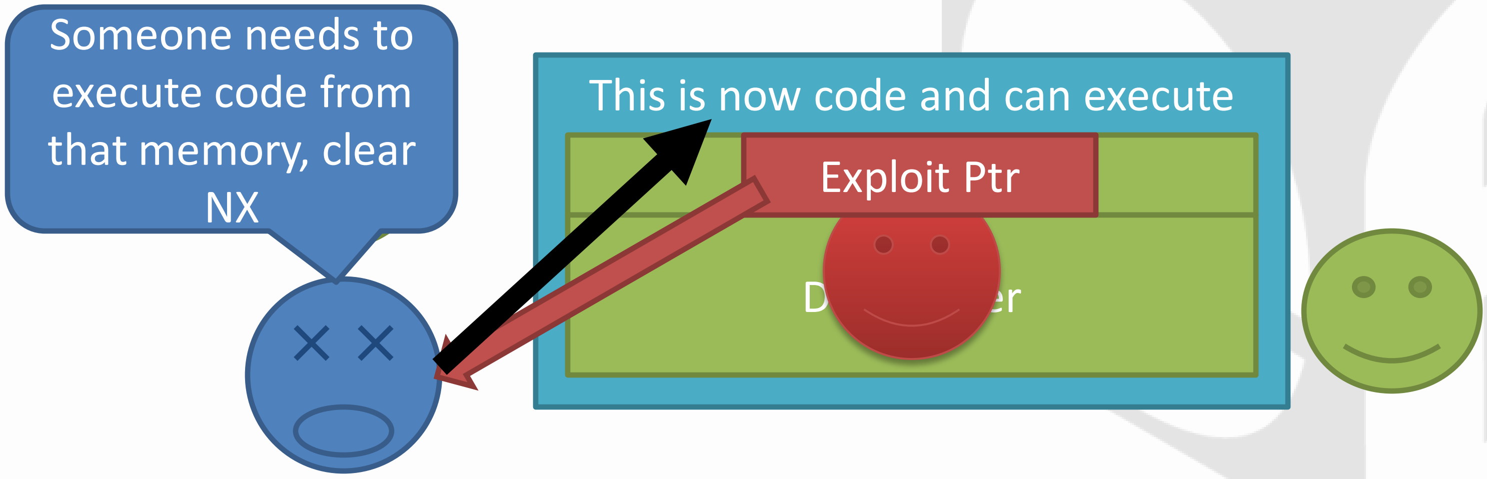
Address Space Layout Randomization (ASLR)

- The altered function pointer is data, so it is referenced to make a call to DXE core without triggering a fault


I need to call that function to do some work…
This is data and cannot execute (NX)
Exploit Ptr
D er

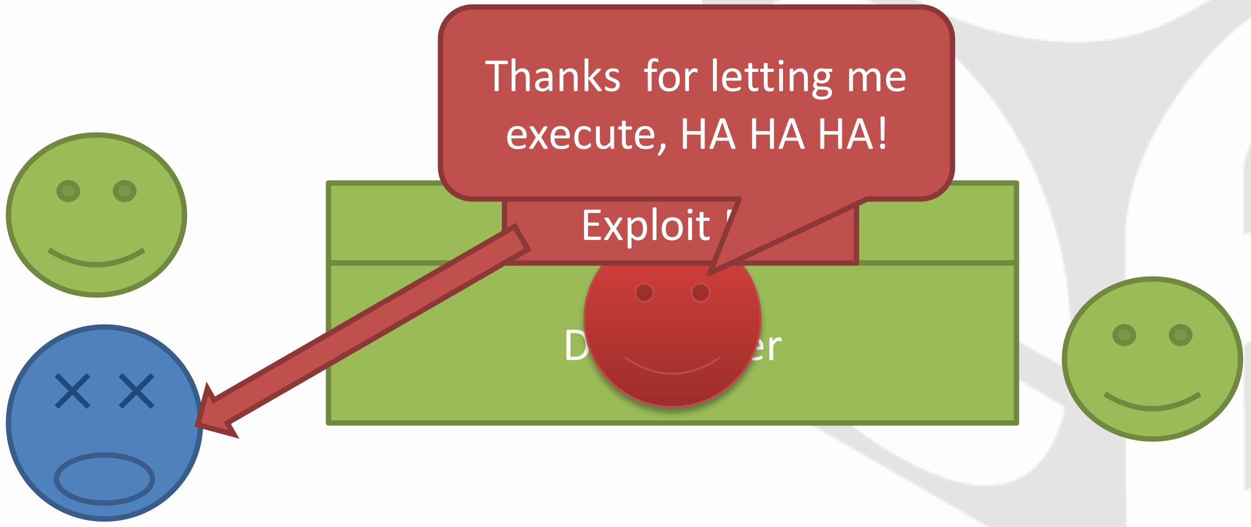
Address Space Layout Randomization (ASLR)

- Legitimate code in DXE core is exploited to disable NX on the memory where the exploit is currently stored


Someone needs to execute code from that memory, clear NX
This is now code and can execute
Exploit Ptr
D er

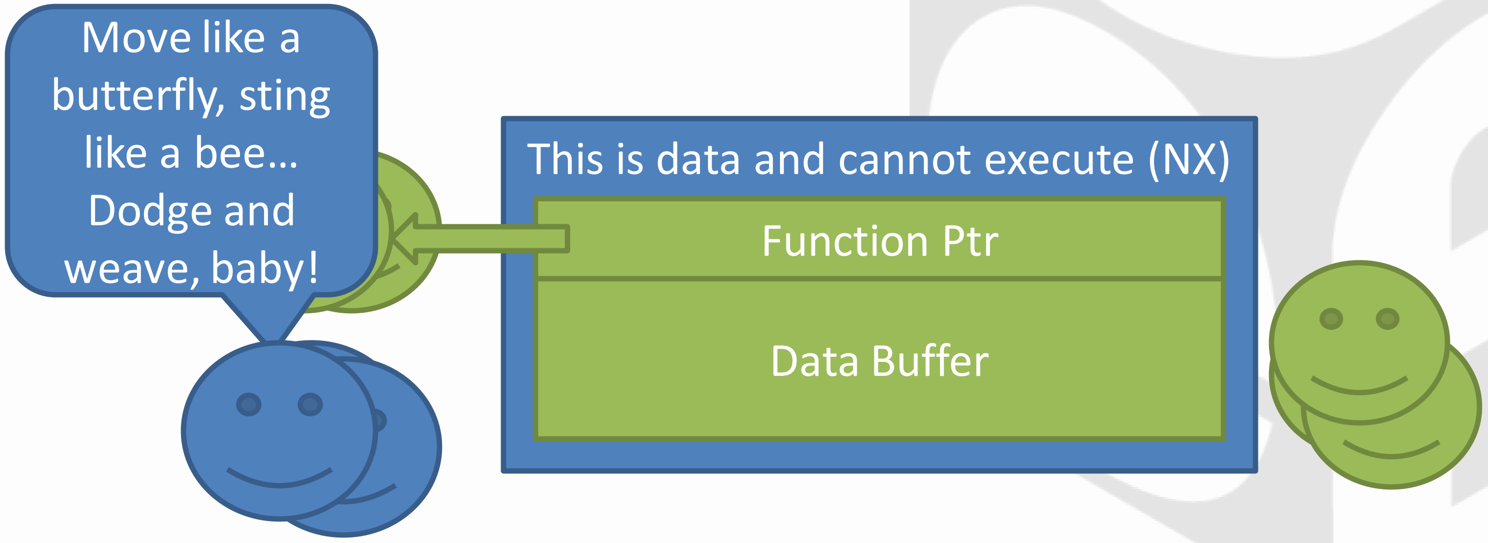
Address Space Layout Randomization (ASLR)

- As a result, the exploit code can now be executed at any time, and NX no longer triggers a page fault



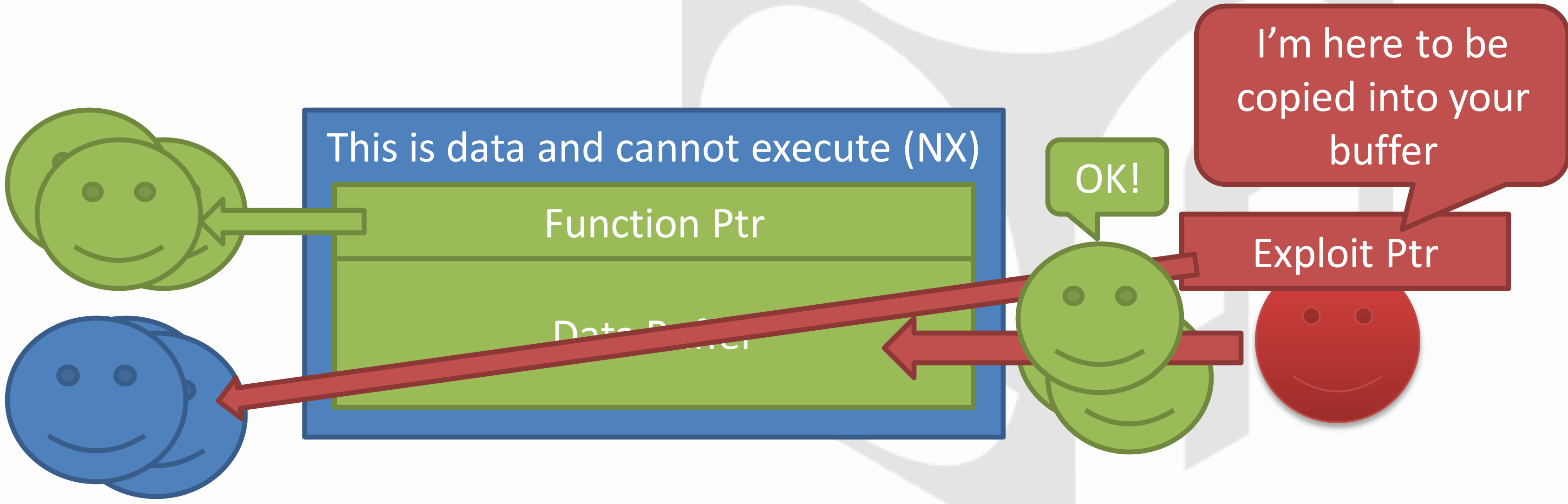
Address Space Layout Randomization (ASLR)

- Address space location randomization causes code to be loaded at different random addresses on every boot


Move like a butterfly, sting like a bee… Dodge and weave, baby!
This is data and cannot execute (NX)
Function Ptr
Data Buffer

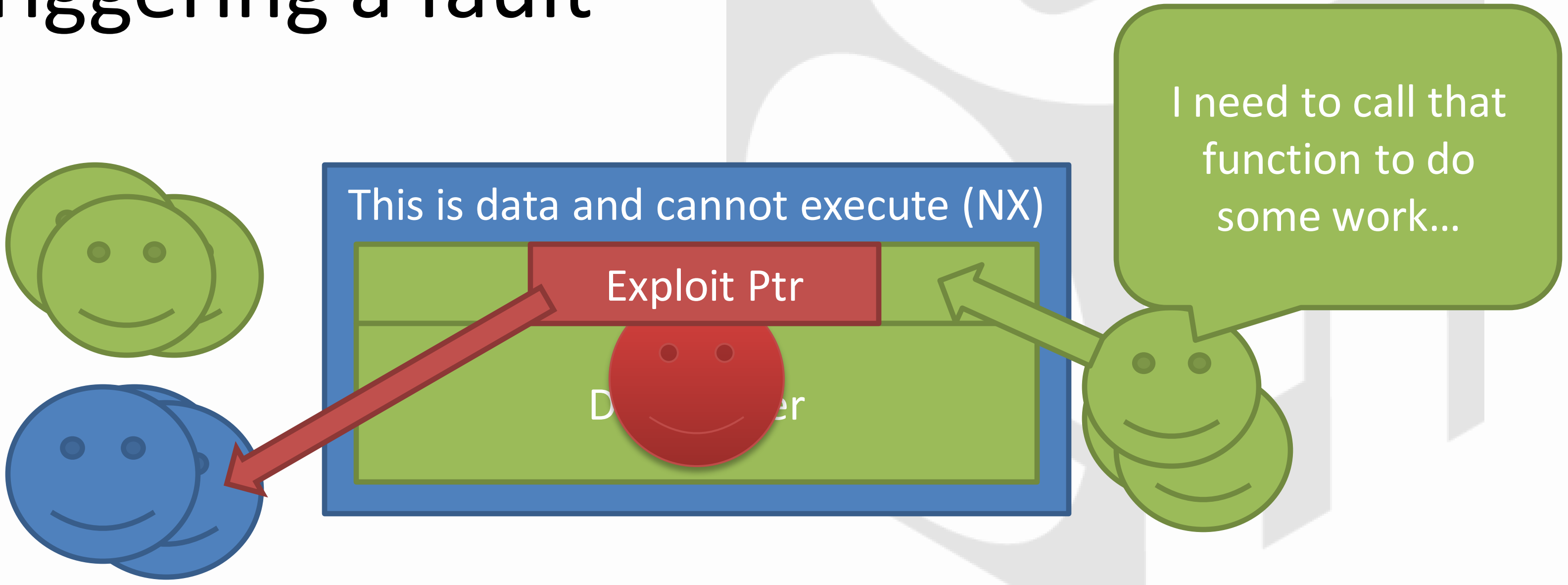
Address Space Layout Randomization (ASLR)

- The exploit can still trick the target into loading it and can change the function pointer to point to a new address


This is data and cannot execute (NX)
Function Ptr
Data Buffer
OK!
I'm here to be copied into your buffer
Exploit Ptr

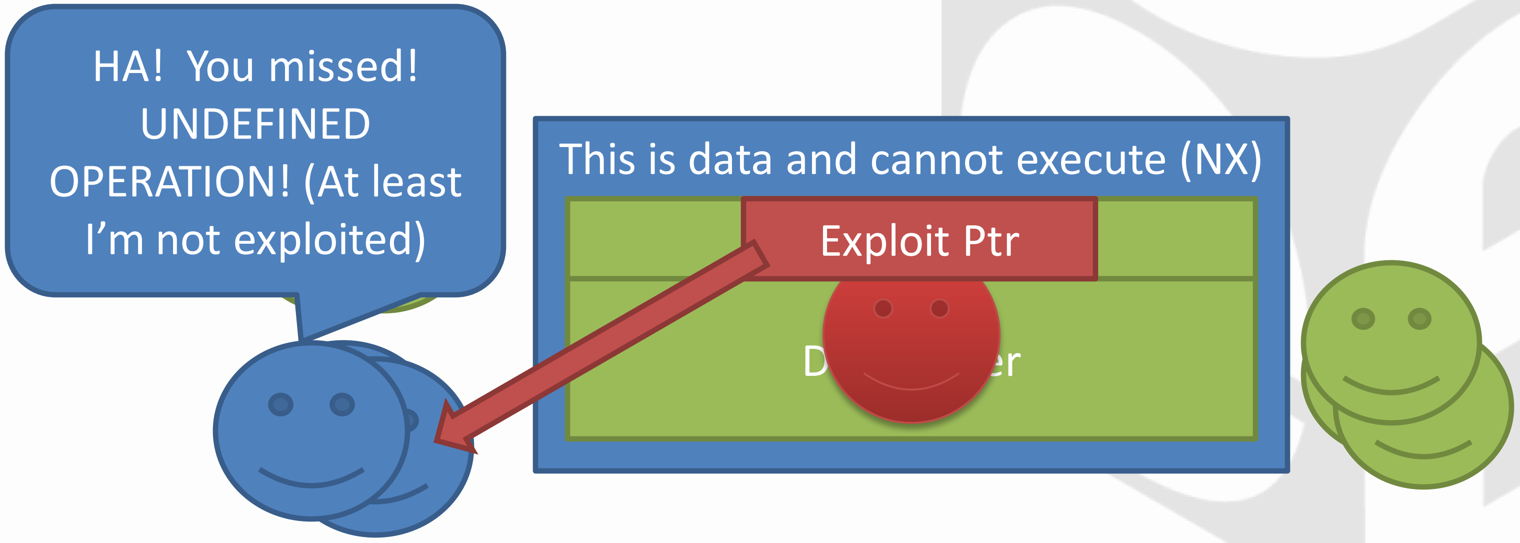
Address Space Layout Randomization (ASLR)

- The altered function pointer is still data, so it is referenced properly without triggering a fault


I need to call that function to do some work...
This is data and cannot execute (NX)
Exploit Ptr
D er

Address Space Layout Randomization (ASLR)

- But the altered pointer can't point to code at a known location, because all code is loaded at random addresses

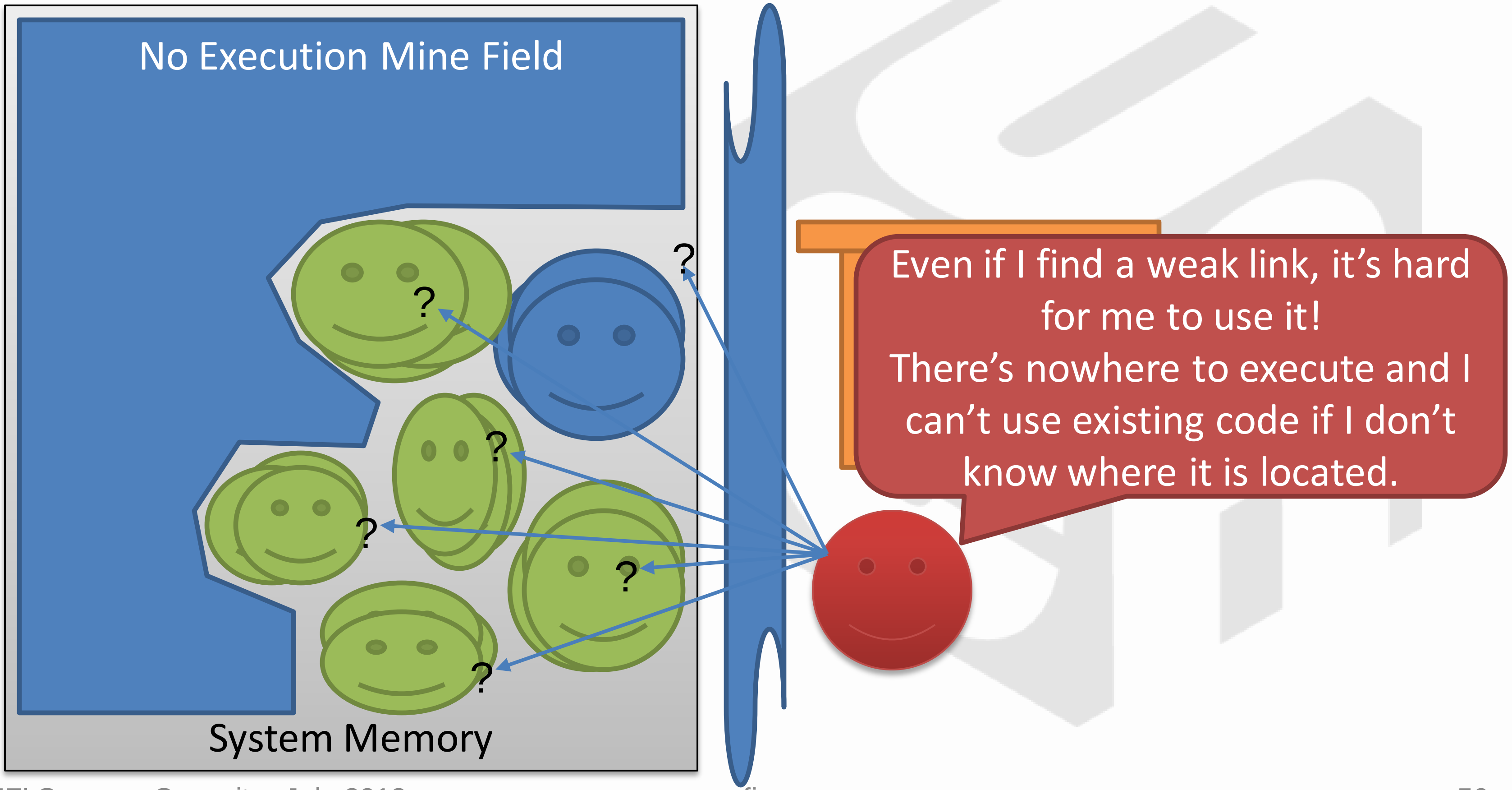

HA! You missed! UNDEFINED OPERATION! (At least I'm not exploited)
This is data and cannot execute (NX)
Exploit Ptr
D
er

Address Space Layout Randomization (ASLR)

- Phoenix ASLR Implementation
  - New BaseRandomNumberLib MdePkg library class
    - Contains random number generation for ASLR
    - Used by PE loaders to randomize load addresses
    - May be replaced to change random number source
  - Enabling PcdAddressSpaceLocRandomizationSupport causes PE loaders to randomize addresses
  - Minimum code alignment can be defined using PcdASLRDefaultAlignmentShift; normally alignment requirements comes directly from PE file format

Conclusion

- Mitigation strategies work together


No Execution Mine Field
?
?
?
?
?
?
System Memory
Even if I find a weak link, it's hard for me to use it!
There's nowhere to execute and I can't use existing code if I don't know where it is located.

Thanks for attending the
UEFI Summer Summit 2012

For more information on the Unified EFI Forum and UEFI Specifications, visit
http://www.uefi.org

presented by