Windows

Killing Uninitialized Memory: Protecting the OS Without Destroying Performance

Joe Bialek (@JosephBialek)
MSRC Vulnerabilities & Mitigations Team

Shayne Hiet-Block (@BlockHiet)
Visual C++ Compiler Team

Microsoft

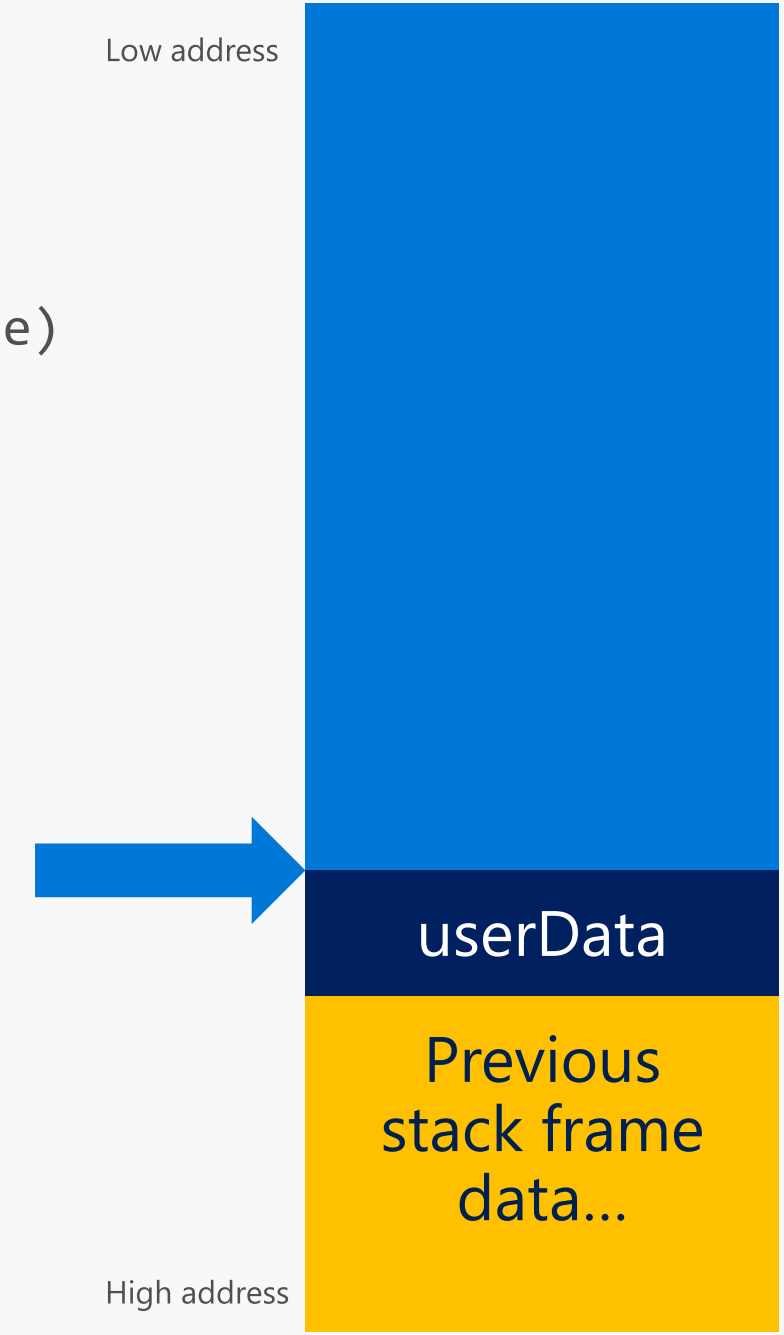
What's The Big Deal?

Circa 2017-2018

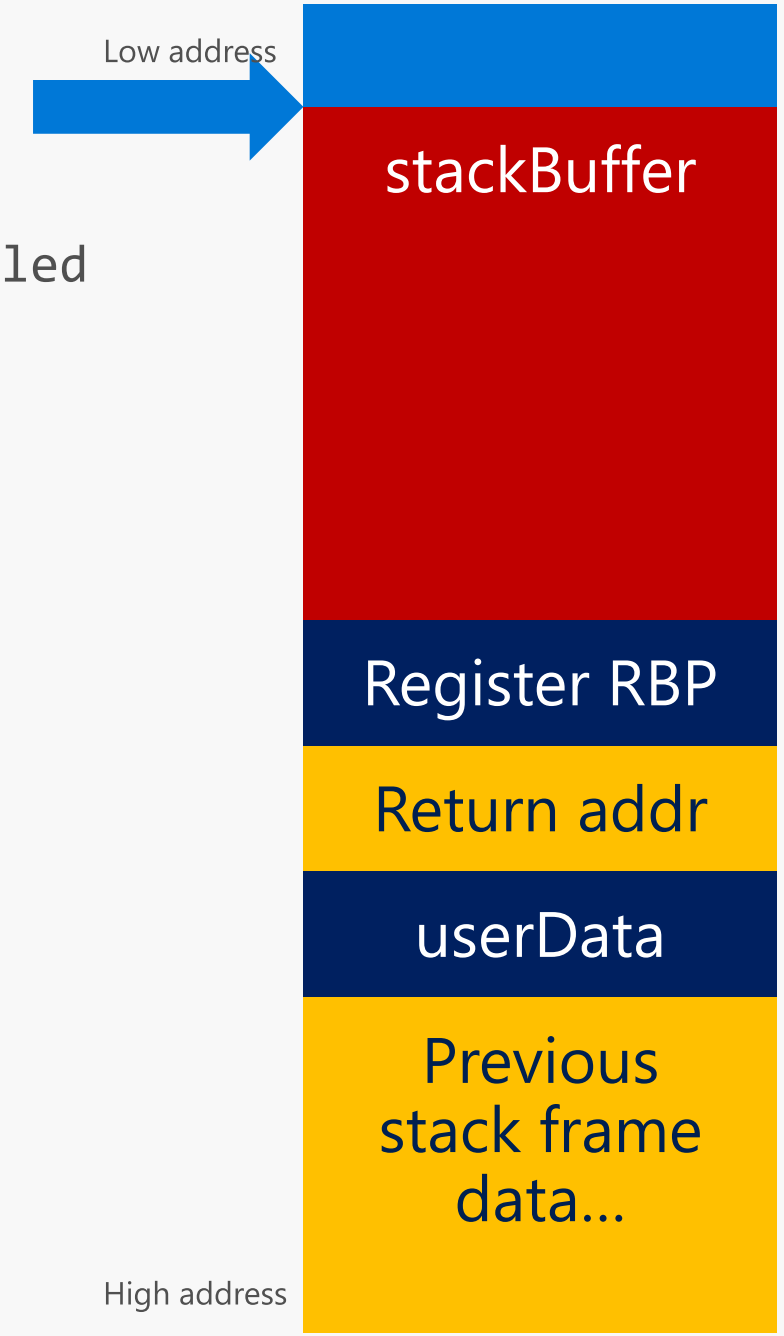
Uninitialized Memory Use

```
int main()
{
    BYTE* userData;

    // get attack-controlled data (from network maybe)
    get_user_data(&userData);

    stack_spray(userData);

    uninit_bug();
}
```

Call this function

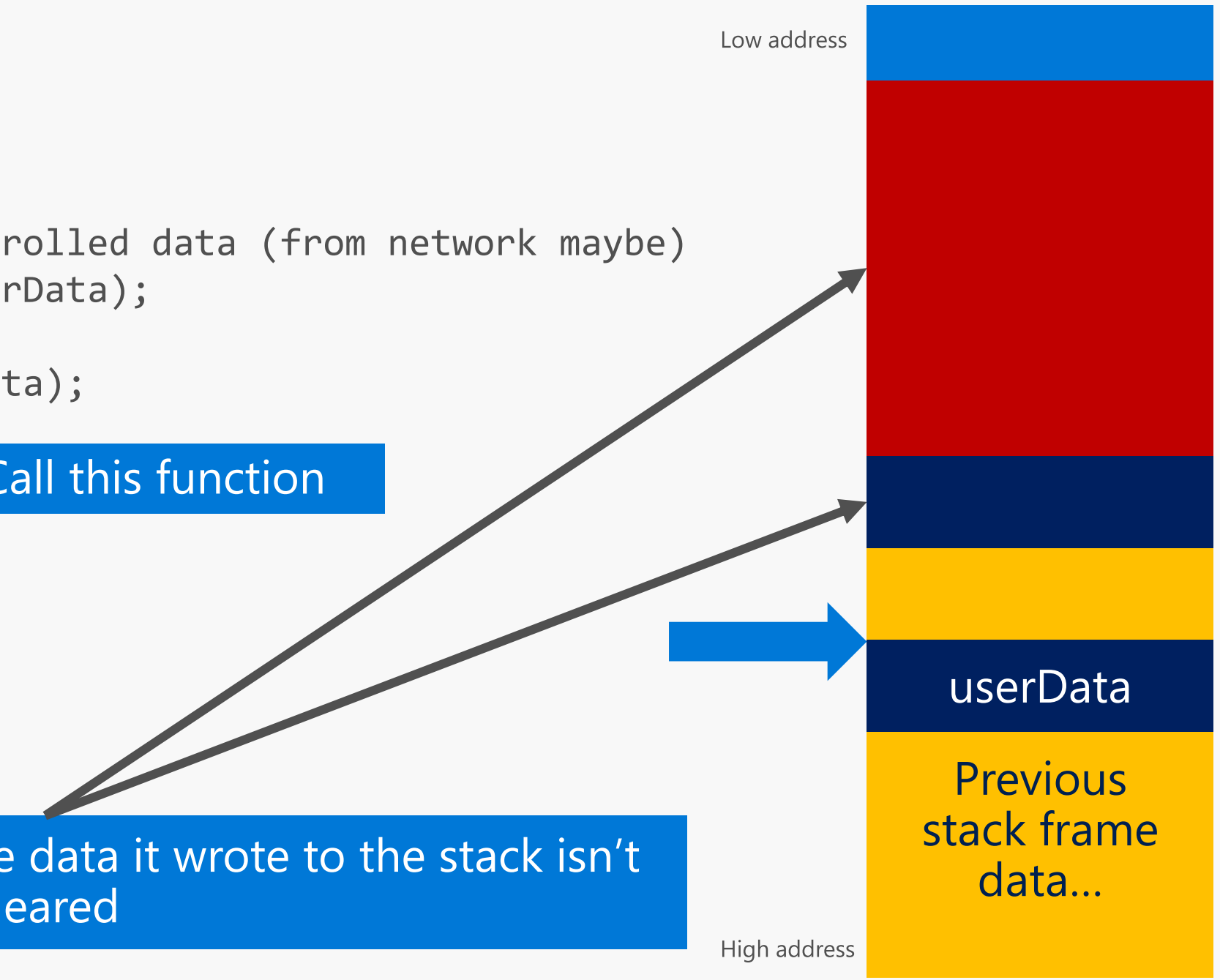


```
void stack_spray(BYTE* userData)
{
    // Assume userData == 1024 bytes long
    // Assume userData contents are attacker controlled

    BYTE stackBuffer[1024];

    memcpy(stackBuffer, userData, 1024);

    DoStuff...
}
```



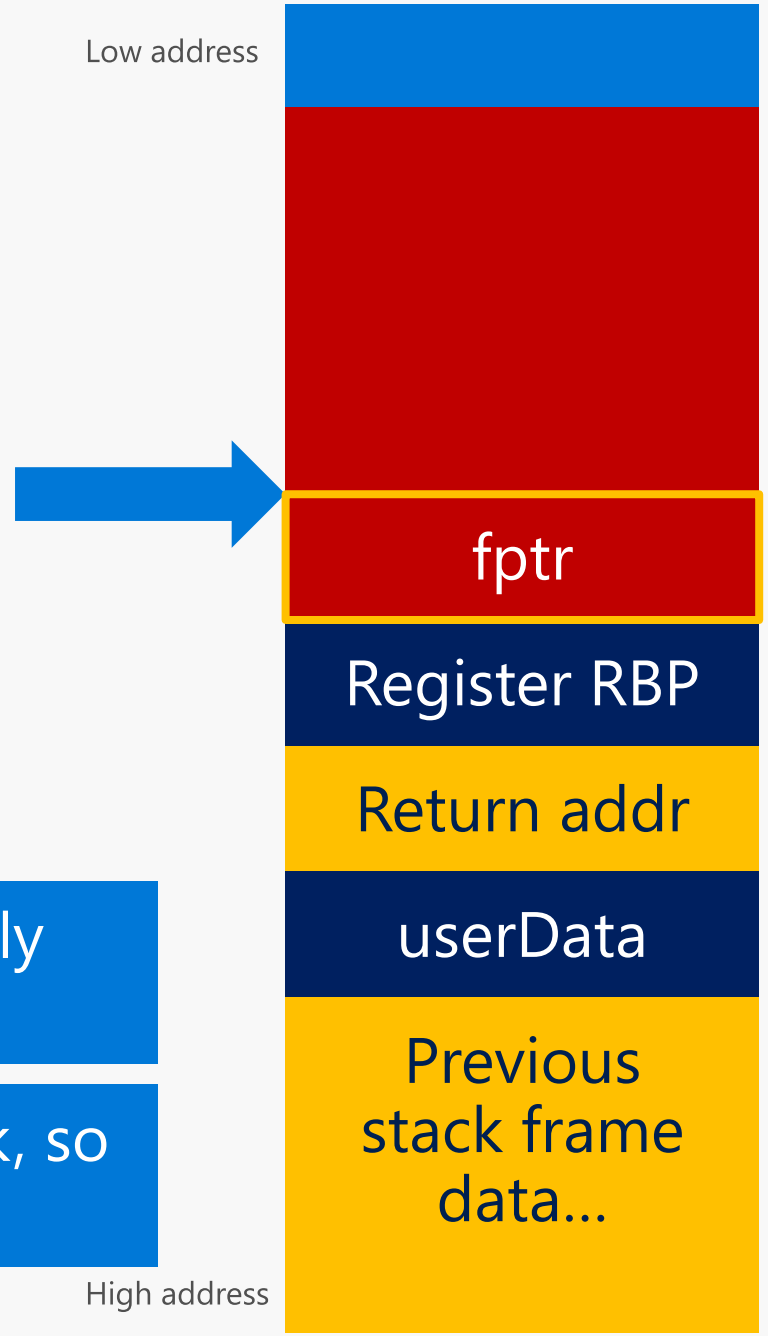
```
int main()
{
    BYTE* userData;

    // get attack-controlled data (from network maybe)
    get_user_data(&userData);

    stack_spray(userData);

    uninit_bug();
}
```

Call this function



stack_spray returns, but the data it wrote to the stack isn't cleared

```
void uninit_bug()
{
    MyFuncPtr fptr; // not initialized

    fptr(); // call function ptr
}
```



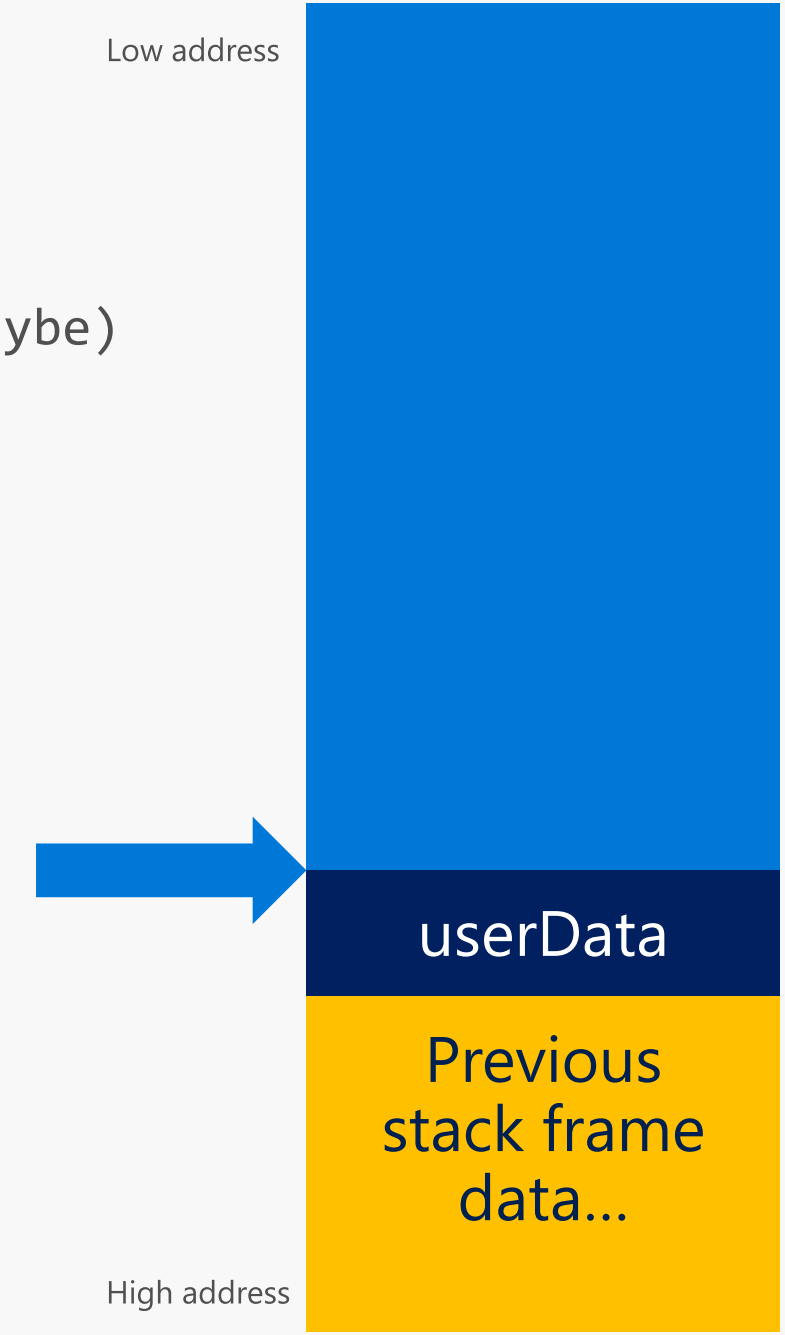
fptr is not initialized, it's "value" is whatever was previously written to this stack address

Attacker-controlled data was previously written to the stack, so attacker-controlled data is used as "fptr"

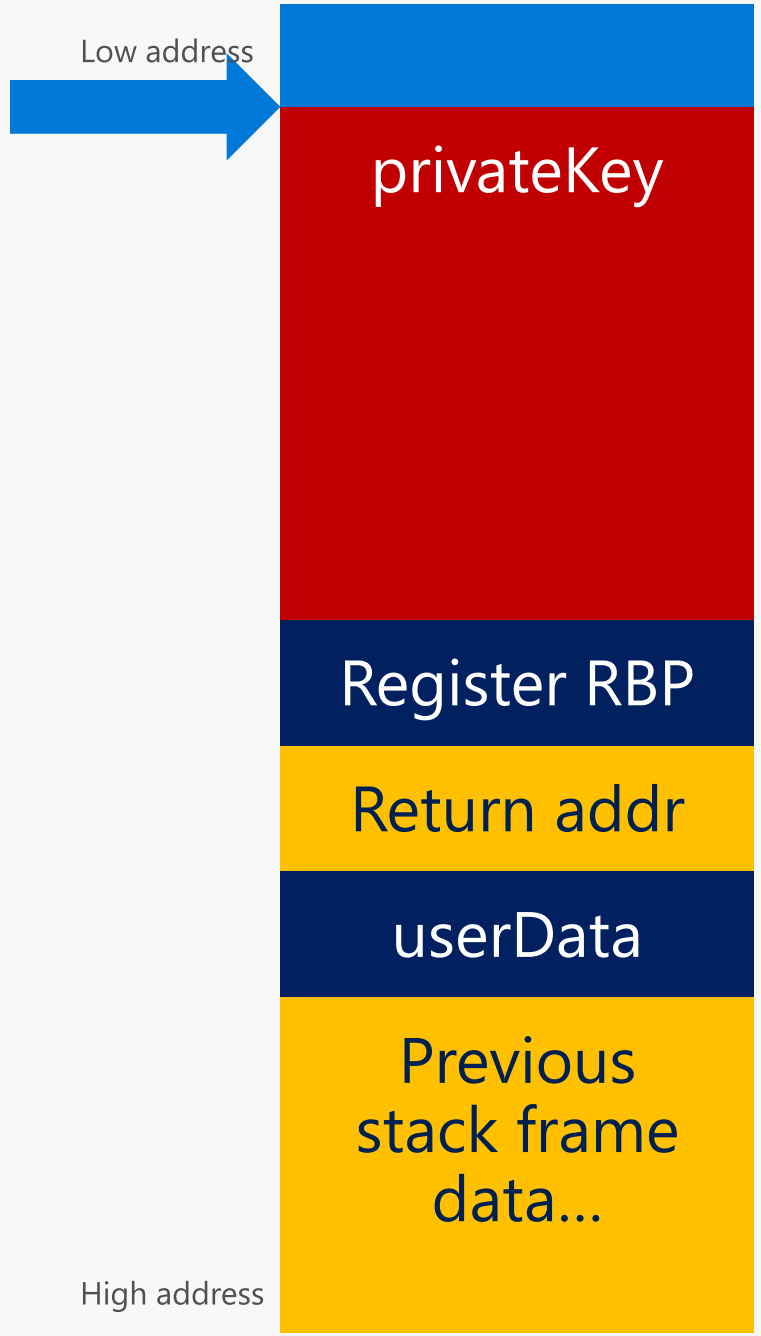
Uninitialized Memory Leak

```
int main()
{
    BYTE* userData;

    // get attacker-controlled data (from network maybe)
    get_user_data(&userData);

    encrypt_buffer(userData);

    send_heartbeat();

    send_heartbeat2();
}
```

Call this function

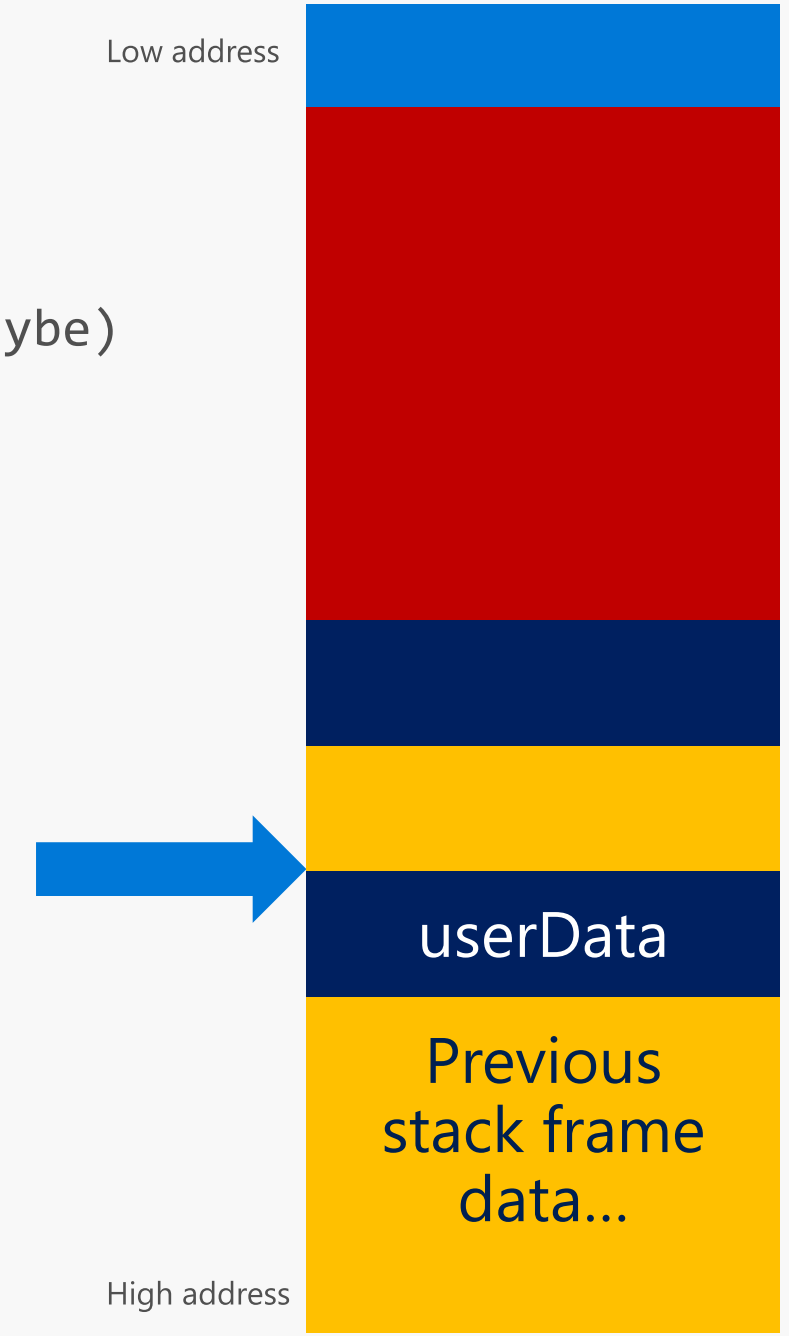


```
int encrypt_buffer(BYTE* buffer)
{
    BYTE privateKey[1024];

    get_encryption_key(&privateKey);

    encrypt_data(privateKey, buffer);

    // Do more stuff…
}
```

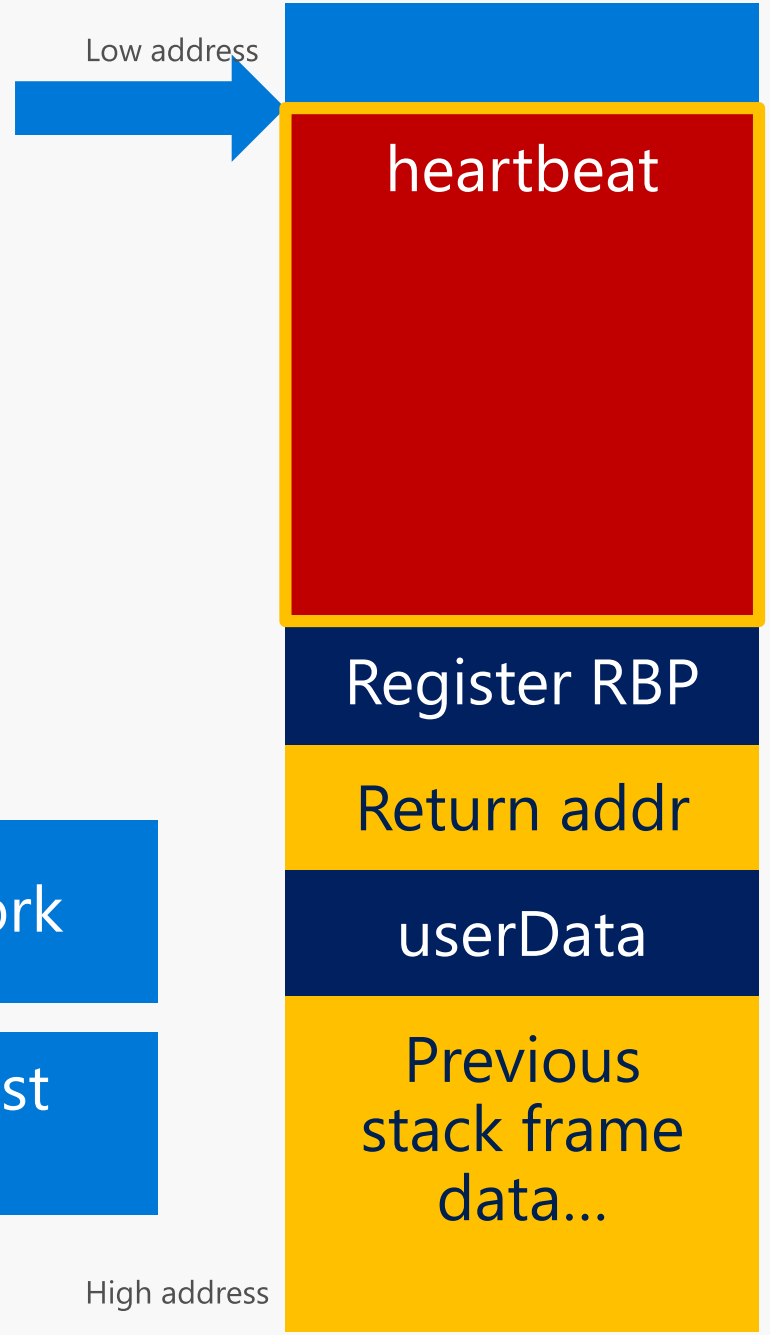


```
int main()
{
    BYTE* userData;

    // get attacker-controlled data (from network maybe)
    get_user_data(&userData);

    encrypt_buffer(userData);

    send_heartbeat();

    send_heartbeat2();
}
```

Call this function



```
int send_heartbeat()
{
    BYTE heartbeat[1024];

    // none of the buffer heartbeat is initialized

    sendToNetwork(heartbeat, sizeof(heartbeat));
}
```

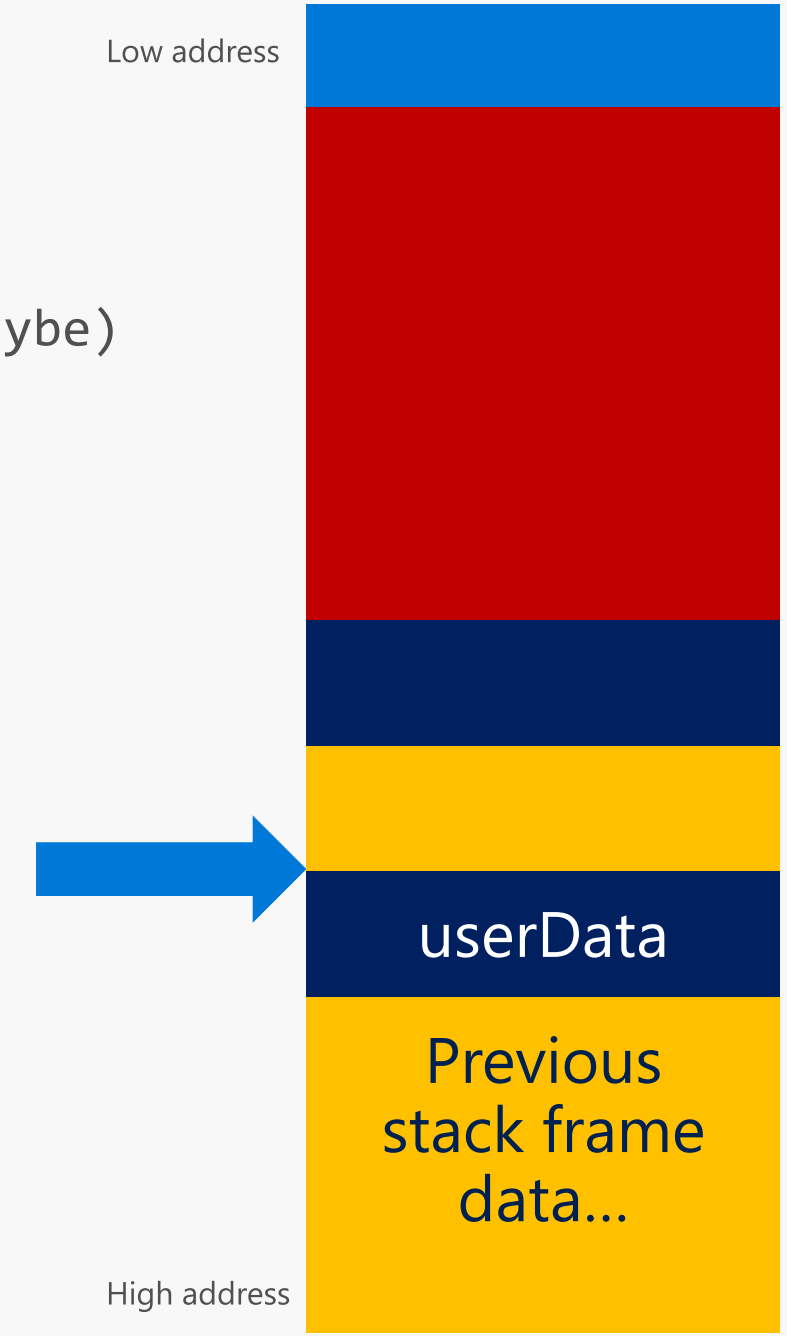


"heartbeat" buffer not initialized and send over the network

Ends up copying the privateKey data since this was the last thing written to this stack address

```
int main()
{
    BYTE* userData;

    // get attacker-controlled data (from network maybe)
    get_user_data(&userData);

    encrypt_buffer(userData);

    send_heartbeat();

    send_heartbeat2();
}
```

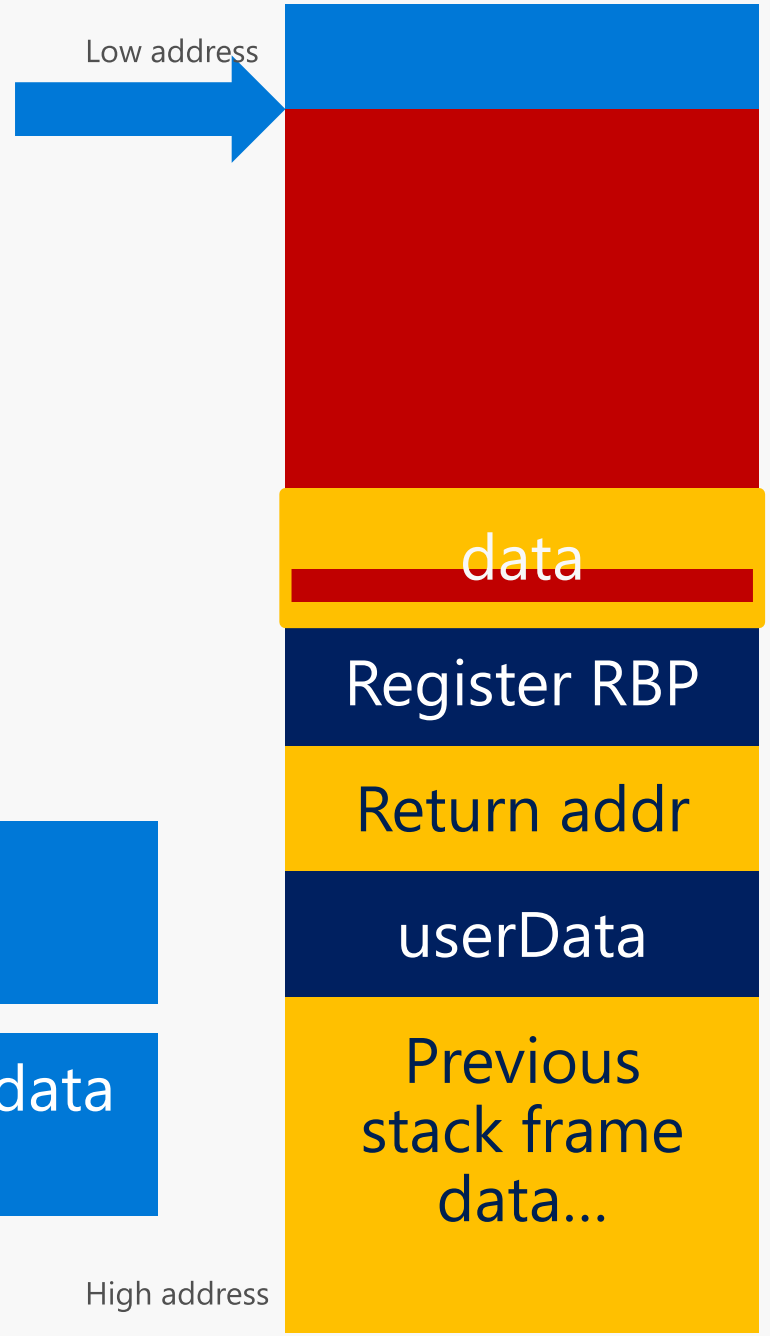
Call this function



```
struct S {
    char field1; // 3 bytes of padding follow this
    int field2;  // offset 0x4 in the structure
};

int send_heartbeat2()
{
    S data = {1, 1};

    sendToNetwork(data, sizeof(data));
}
```

Padding between field1 and field2 is not initialized

Copying the padding field ends up copying the privateKey data that was previously written to this address



Examples not comprehensive, but represent real-world scenarios

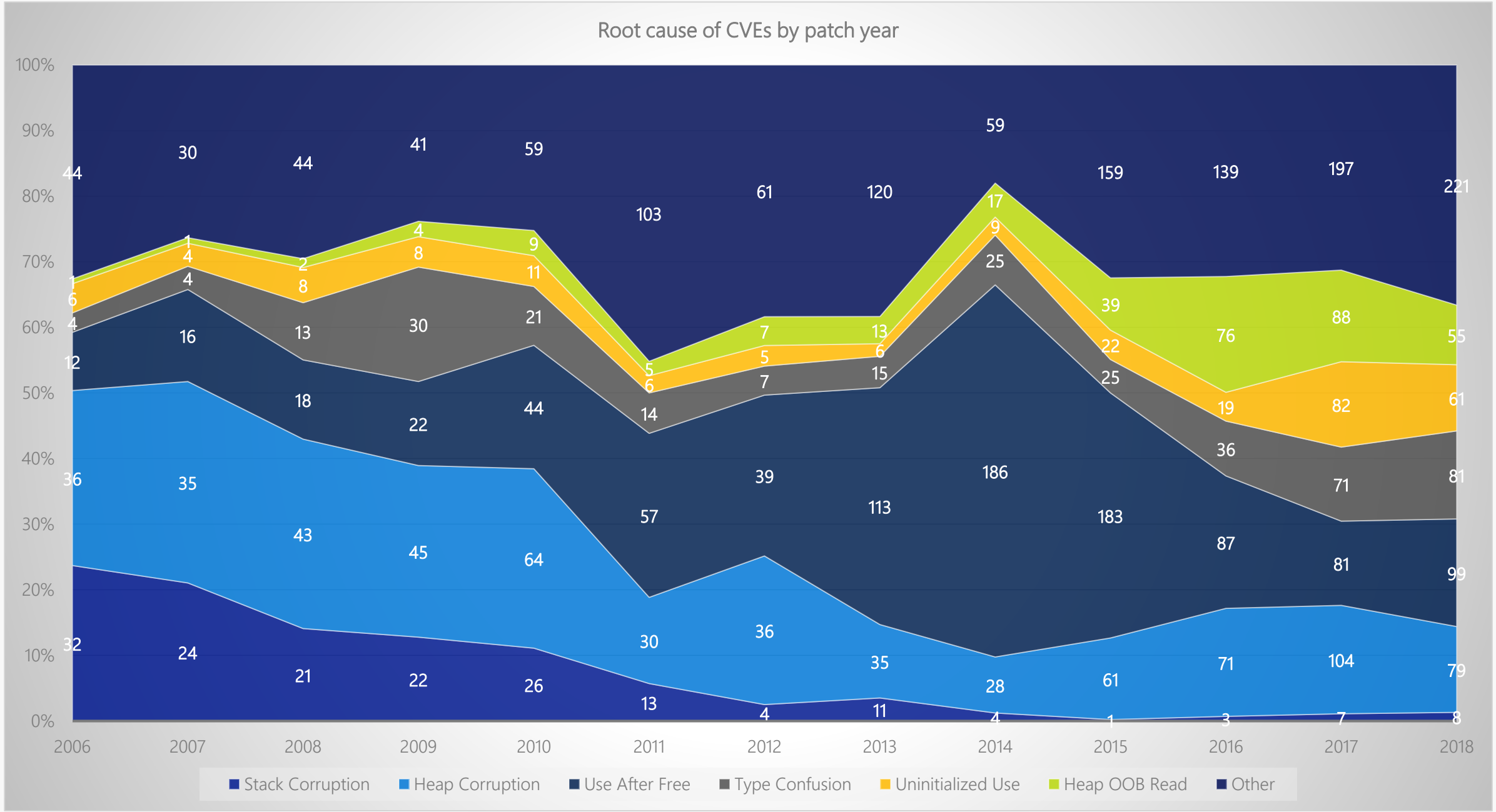
Uninitialized array size, array index, bool's, etc., can all lead to security issues

Many real-world examples of leaking targeted data, or influencing uninitialized values

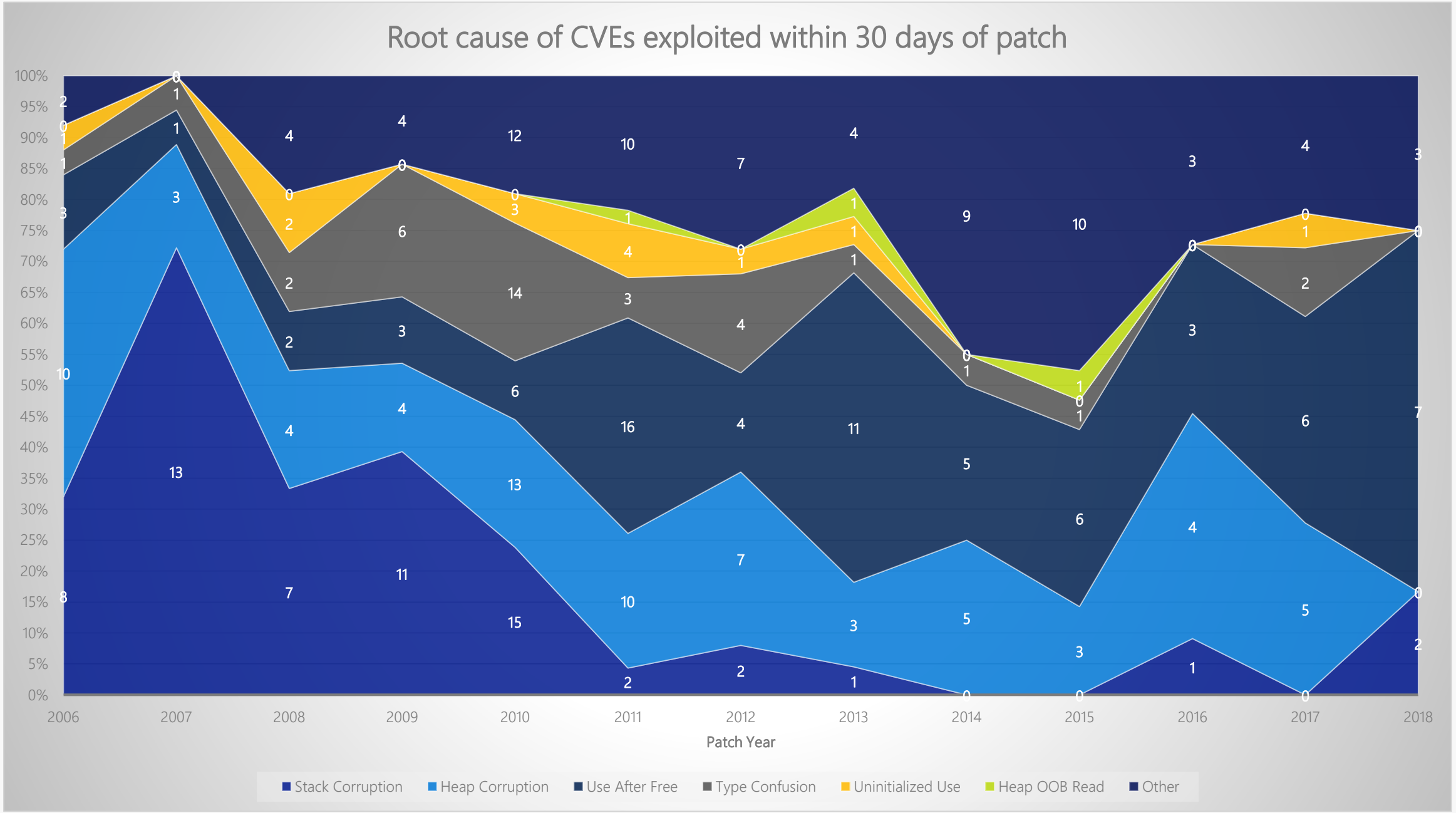
Remember Heartbleed?

Software vendors must be conservative & fix bugs

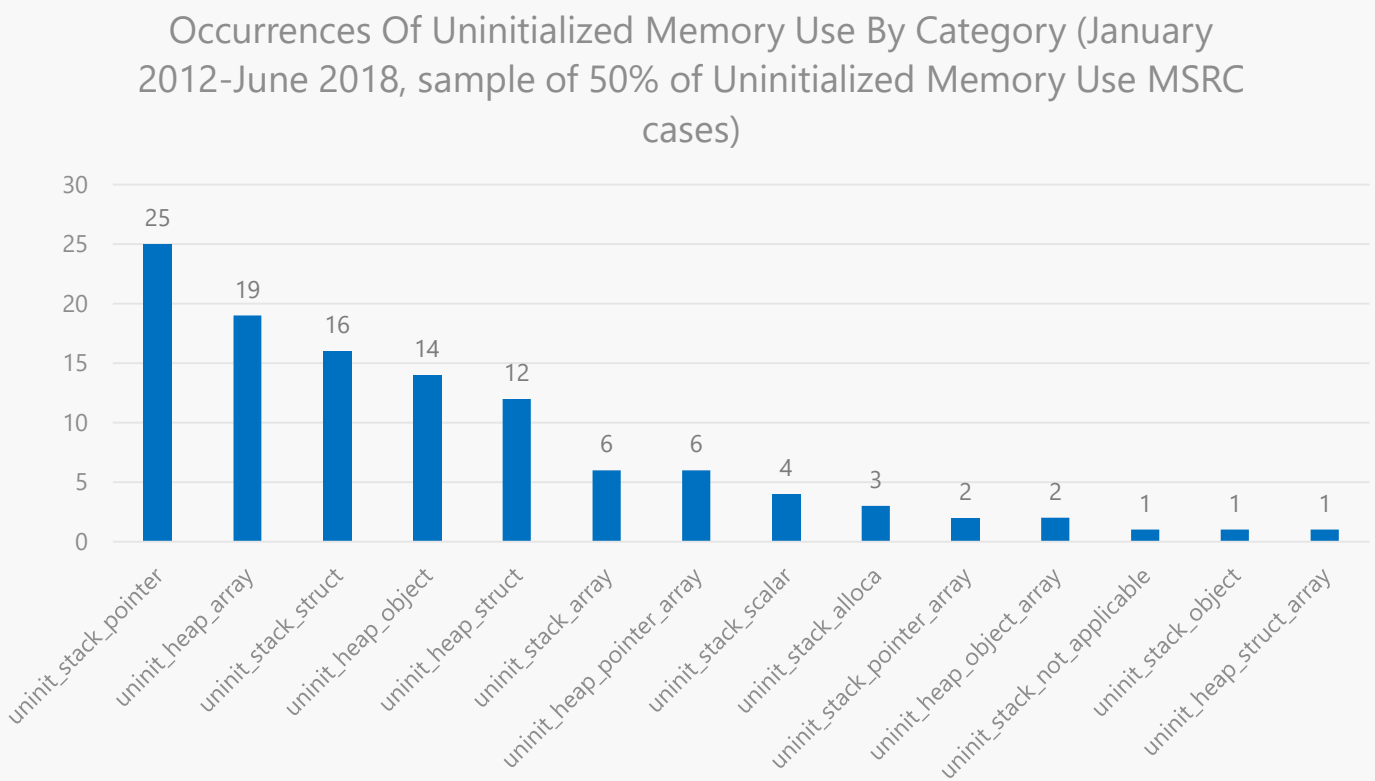
Extremely difficult to prove an uninitialized memory bug has no security impact

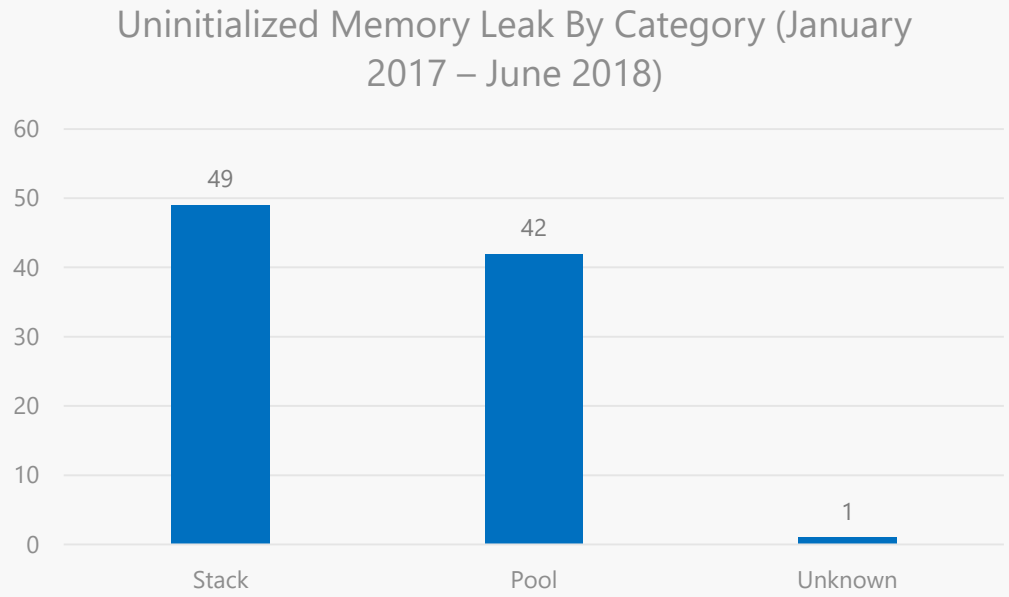


Note: Uninitialized Use include Uninitialized Leak on this graph



Note: Uninitialized Use include Uninitialized Leak on this graph





This graph contains detailed root cause information for roughly 50% of MSRC cases between January 2012 – June 2018 that root cause to "Uninitialized Memory Use". Super detailed information necessary to create these fine-grained statistics is not available for all MSRC cases dating back to 2012.

Let’s Do Something!

What Isn't Working?

Static Analysis	• Developer's don't have to use it • Can't find everything
Code Review	• Not possible to review everything • Code reviewed on 32-bit might have been okay, but moving to 64-bit introduced new padding
Fuzzing	• Hard to find uninitialized memory leaks (no crash) • Impossible to get 100% test coverage

Windows must run fast, run all existing code, and be extensible for all future changes 30+ years from now

Security	Does the mitigation make things safer?
ABI	Cannot be broken
Performance	Programs & the compiler must be fast & small
Compatibility	All existing programs must continue working
Developer Experience	Developers must remain productive
Flexible	Must not paint us into a corner

Find secure solutions, then iterate & refine to satisfy all criteria

Observations

Stack variables make up over 50% of all uninitialized memory bugs

Initializing at declaration immediately mitigates most uninitialized memory bugs

C++ Core Guidelines recommends initializing stack variables at declaration

ES.20: Always initialize an object

Reason Avoid used-before-set errors and their associated undefined behavior. Avoid problems with comprehension of complex initialization. Simplify refactoring.

InitAll

Automatically initialize stack variables to zero

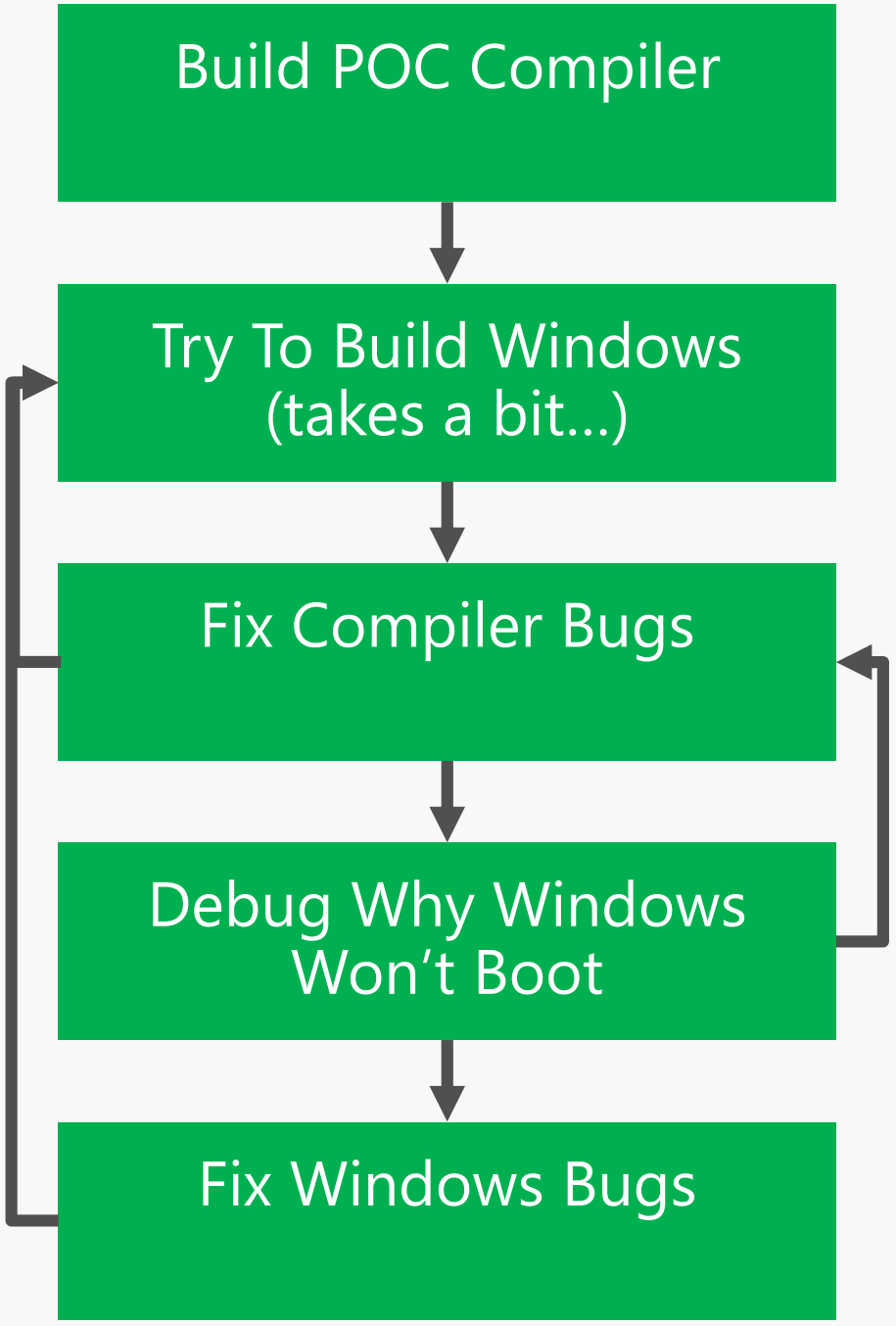
Zero is the safest value from security point-of-view

Implemented in front-end, looks identical to zero-initialization in source code

What might happen if we automatically initialize variables?

Performance	Stack frame already in the L1 cache, accesses should be fast
	Hopefully, initializations proven away by the compiler
	Compiler throughput not affected
ABI	Not affected
Compat, Flexibility	Should not be affected
Developer Experience	Will discuss later, should not be an issue

How we prototyped InitAll



Due to buggy code, force initializing non-POD classes not easily possible

Settled on POD structures, arrays, scalars

Got Windows booting, the idea isn't completely crazy!

Performance Testing

Performance Testing (not exhaustive list)

Server Scenarios	• Virtualization Storage & Networking • Container Perf (Hyper-V and Windows Server) • Native Networking • Local/Remote Storage (RAM disk) • Web Server
Client Scenarios	• Browser • Boot, hibernate, restore • UI responsiveness

Performance Targets

Synthetic Tests	Under 2% regression
Real-World Tests	Under 1% regression
Code Size	Under 1% increase

Certain tests (like web server) are strong indicators of overall system perf

Don’t have test coverage for everything, need to focus on tests that highlight overall system perf

Initial Results

Results	Most scenarios regressed 3-4% Hyper-V Containers regressed ~10%
Observations	10% regression – Caused by a single stack variable, code path only used during performance testing Arrays – Difficult to optimize away initialization if array is later set using a loop Most kernel-bugs are in POD structures
Tweaks	Only apply InitAll to POD structures Opt-out variable causing 10% performance hit

Performance Analysis After Tweaks

- Some tests slightly above our performance targets
- Code size increase: Around 0.1% (totally fine)
- Start analyzing WPA traces

How Perf Issues Can Manifest

New initializations scattered through code leads to many more initializations in aggregate (hard to diagnose)

New initializations == bigger functions == less inlining

Hot loops perf could be destroyed if variables are declared in the loop

Calling memset incurs a much higher performance penalty than memset's that can be unrolled

Strategy

Focus on reducing total # of initializations

Reduces impact to inlining

Reduces “general slowdown” due to more initializations in aggregate

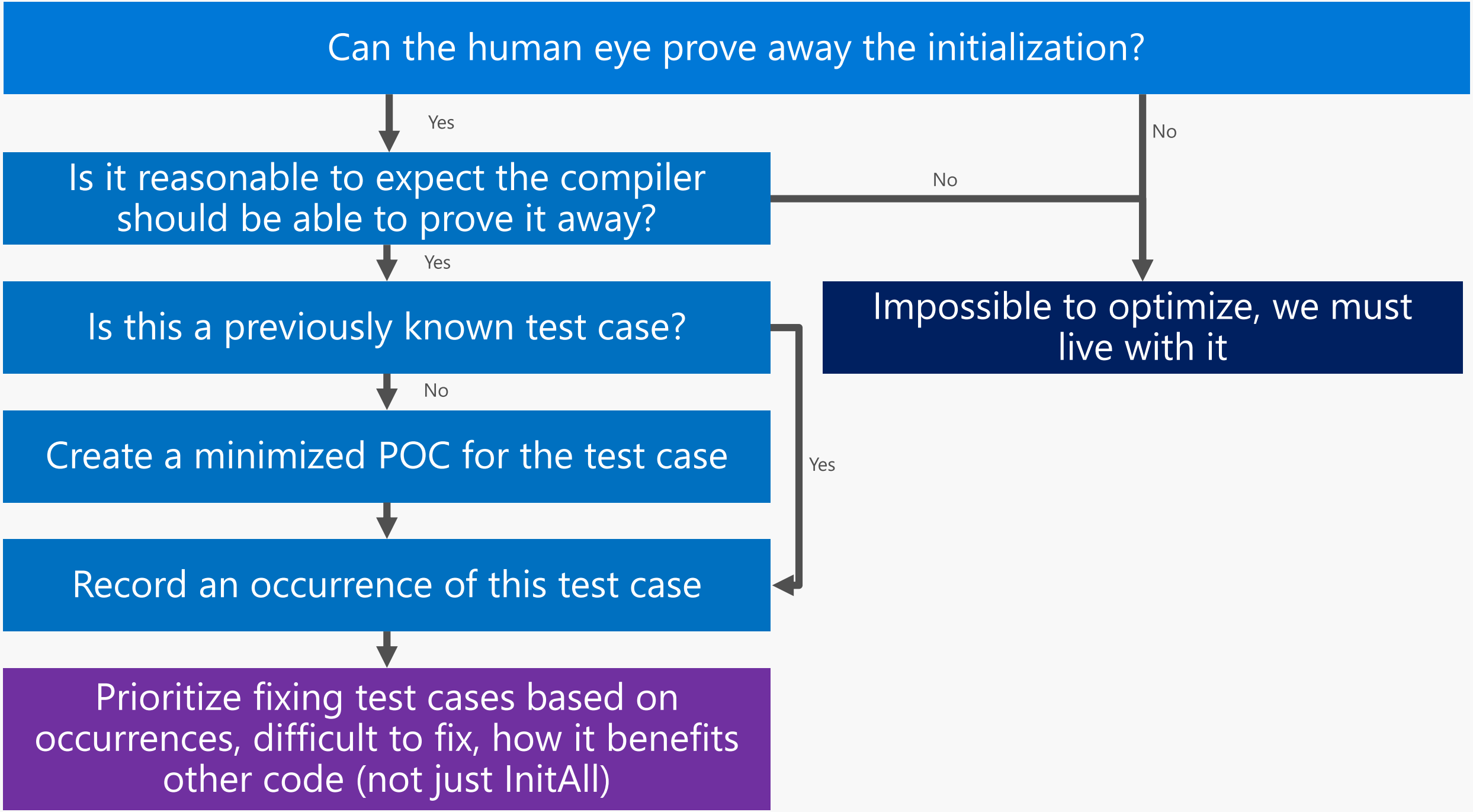
Analysis Methodology

Run perf tests and collect Windows Performance Analyzer traces

Disassemble hot functions (with InitAll and without) using IDA Pro

Use IDA Pro script to count # of stores/loads and calls to memset

Manual analysis of functions with large diffs


Can the human eye prove away the initialization?
Yes
No
Is it reasonable to expect the compiler should be able to prove it away?
No
Yes
Impossible to optimize, we must live with it
Is this a previously known test case?
Yes
No
Create a minimized POC for the test case
Record an occurrence of this test case
Prioritize fixing test cases based on occurrences, difficult to fix, how it benefits other code (not just InitAll)

Compiler Optimizations

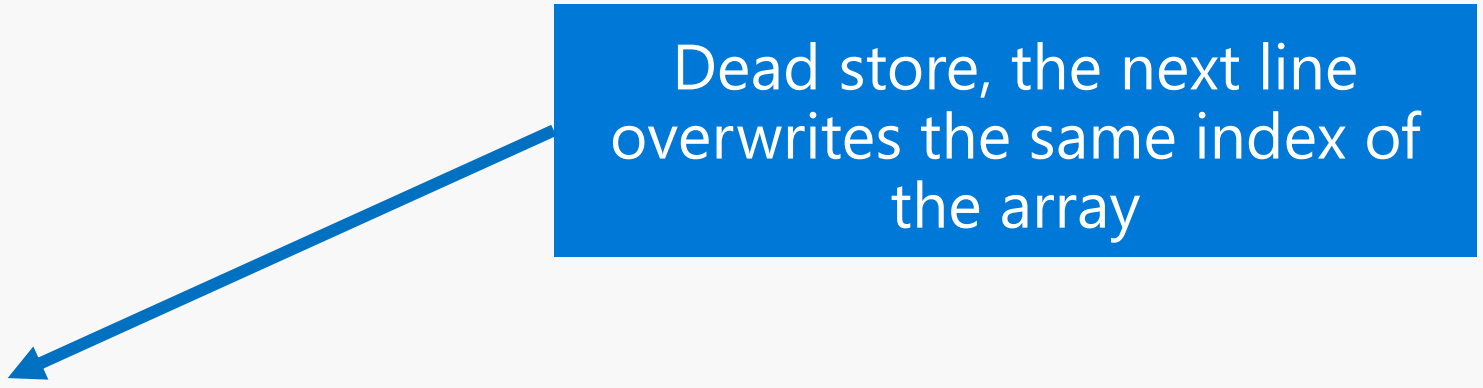
Basic Dead Store Example

```
void foo()
{
    int x = 0;
        x = 2;

    TakeAddressOf(&x);
}
```

Optimizes correctly already

Basic Dead Store Example

```
void foo()
{
    int arr[10];
    int x = 1;
    int y = 1;
    arr[x] = 5;
    arr[y] = 6;
}
```



Optimizes correctly already

Current examples are optimized with “dead store elimination” optimizer

Requires other additional optimizations to run first

20+ years old, hard to extend

New “Redundant Store Elimination” optimizer created

Uses SSA, Value Numbering, & other tricks

Easier to extend to more complicated patterns

SSA

The declare-at-initialization of “x” is redundant, can SSA prove this?

“x” is always re-set before use

```
int x = 0;
int y = 1;

if (rand()) {
    x = 3;
    y = x;
} else {
    x = 2;
}

… = x;
```

SSA

Think of SSA like a versioning system

Every definition is assigned a unique number

Every “use” references the number

Join points (multiple possible values) form a “PHI”

Each possible value listed in the PHI

```
int x<*1> = 0;
int y<*2> = 1;

if (rand()) {
    x<*3> = 3;
    y<*4> = x<3>;
} else {
    x<*5> = 2;
}
x<*6> = PHI{x<3>,x<5>}
… = x<6>;
```

SSA

x<1> is not read anywhere

The assignment to <1*> can be removed

SSA has other uses, like propagating values

SSA is run at multiple points during compilation

SSA is fast and scales well for what it provides

```
int x<*1> = 0;
int y<*2> = 1;

if (rand()) {
    x<*3> = 3;
    y<*4> = x<3>;
} else {
    x<*5> = 2;
}
x<*6> = PHI{x<3>,x<5>}
… = x<6>;
```

Value numbering

```
int x<*1> = 1<1>;
int y<*2> = 1<1>;
```

Assign a number to all initial values

```
int arr<*3><2>[10];
arr<3>[x<1>] = 5<3>;
arr<3>[y<2>] = 6<4>;
```

Value numbering

```
int x<*1><1> = 1<1>;
int y<*2><1> = 1<1>;
```

Propagates values through SSA

```
int arr<*3><2>[10];
arr<3><2>[x<1><1>] = 5<3>;
arr<3><2>[y<2><1>] = 6<4>;
```

Value numbering

```
int x<*1><1> = 1<1>;
int y<*2><1> = 1<1>;
```

Every expression given a value

Both store addresses have the same number

```
int arr<*3><2>[10];
arr<3><2>[x<1><1>]<5> = 5<3>;
arr<3><2>[y<2><1>]<5> = 6<4>;
```

```
void foo()
{
    int x = 0;
    bar();
    x = 2;
    TakeAddressOf(&x);
}
```

Why is this not being optimized away?

Interference Analysis

"Could variable 'x' be killed between point A and point B?"

Interference Analysis attempts to answer this question!

In MSVC, "Interf" runs early and uses variable scope for locals

Interference Analysis

```
void foo()
{
    int x = 0;
    bar();
    x = 2;
    TakeAddressOf(&x);
} // x live until here
```

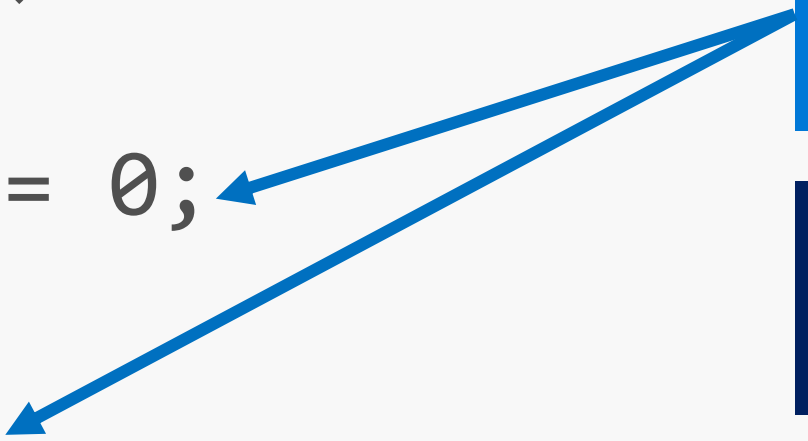
Interf: “x” is untracked (it’s address is passed to some unknown function)

It could be used/modified by code outside of this function

“x” is “untracked” for this entire scope

Interference Analysis

```
void foo()
{
    int x = 0;
    bar();
    x = 2;
    TakeAddressOf(&x);
} // x live until here
```



Optimizer: “Is there interference on ‘x’ between these two points?”

Interf: “Yes, bar() could use or modify ‘x’ because ‘x’ is untracked”

Optimizer: “There is interference so I must keep both stores”

Human: “Yeah but ‘x’ doesn’t escape the function until after the call to bar()”

On-Demand Flow Based Interference

```
void foo()
{
    int x = 0;
    bar();
    x = 2;
    TakeAddressOf(&x);
} // x live until here
```

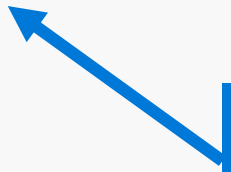
RSE-Specific Optimization (modifying Interf globally is hard)

Understands at what point ‘x’ escapes

bar() cannot interfere with ‘x’ because ‘x’ has not yet escaped

First store is now eliminated as expected, function optimized correctly

```
void MemsetEliminationTest() {
    MyStruct s;
    memset(&s, 0, sizeof(s));

    while (GetUnknown())
    {
        if (GetUnknown()) {
            return;
        }
    }

    memset(&s, 0, sizeof(s));
    TakeAddressOf(&s);
}
```



Why doesn't this initialization get removed?

This is almost the same pattern as before, but there's a loop in between, right?

```
void MemsetEliminationTest() {
    MyStruct s;
    memset(&s, 0, sizeof(s));

    while (GetUnknown())
    {
        if (GetUnknown()) {
            return;
        }
    }

    memset(&s, 0, sizeof(s));
    TakeAddressOf(&s);
}
```

Loop has multiple escape points

Analysis was upgraded to handle this properly

With upgraded analysis, this memset is removed

Shouldn’t the first initialization be removed, not the second?

```
void MemsetEliminationTest() {
    MyStruct s;
    memset(&s, 0, sizeof(s));

    while (GetUnknown())
    {
        if (GetUnknown()) {
            return;
        }
    }

    memset(&s, 0, sizeof(s));
    TakeAddressOf(&s);
}
```

Forward pass: Scan forwards and check if another initialization makes this one redundant

Forward pass misses

```
void MemsetEliminationTest() {
    MyStruct s;
    memset(&s, 0, sizeof(s));

    while (GetUnknown())
    {
        if (GetUnknown()) {
            return;
        }
    }

    memset(&s, 0, sizeof(s));
    TakeAddressOf(&s);
}
```

Backwards pass: Scan backwards and check if this initialization is a duplicate of one that already happened

Backwards pass succeeds and removes the second initialization

```
void MemsetEliminationTest() {
    MyStruct s;
    memset(&s, 0, sizeof(s));

    while (GetUnknown())
    {
        if (GetUnknown()) {
            return;
        }
    }

    memset(&s, 11, sizeof(s));
    TakeAddressOf(&s);
}
```

Backwards pass is only proving the second memset is the same as the first

If the second memset uses a different fill pattern, neither will be removed

```
void MemsetEliminationTest2()
{
    MyStruct s;
    memset(&s, 0, sizeof(s));

    for (int i = 0; i < GetUnknown(); i++) {
        if (GetUnknown()) {
            continue;
        }

        memset(&s, 0, sizeof(s));
        TakeAddressOf(&s);
    }
}
```

Why doesn't this initialization get removed?

Every iteration of the loop re-initializes the structure

```
void MemsetEliminationTest2()
{
    MyStruct s;
    memset(&s, 0, sizeof(s));

    for (int i = 0; i < GetUnknown(); i++) {
        if (GetUnknown()) {
            continue;
        }

        memset(&s, 0, sizeof(s));
        TakeAddressOf(&s);
    }
}
```

Flow Sensitive Logic

‘s’ becomes untracked

GetUnknown can be called while ‘s’ is untracked (e.g. second loop iteration)

Memset cannot be eliminated because when GetUnknown is called, ‘s’ is untracked

Requires path-sensitive analysis, not just flow sensitive

```
void MemsetEliminationTest2()
{
   MyStruct s;
   memset(&s, 0, sizeof(s));

   for (int i = 0; i < GetUnknown(); i++) {
      if (GetUnknown()) {
         continue;
      }

      memset(&s, 0, sizeof(s));
      TakeAddressOf(&s);
   }
}
```

Path sensitive is harder and often doesn't get the gains you hope it will

We didn't optimize this example, hopefully perf is okay ☺

```
void PartialMemsetEliminationTest1()
{
    MyStruct s;
    memset(&s, 0, 24);

    s.Field1 = 1;
    // No init for fields 2 - 5
    s.Field6 = 6;

    TakeAddressOf(&s);
}
```

Assume MyStruct has 6 fields, each 4 bytes

Re-initialize first and last field

Couldn’t we make the memset a bit smaller?

```
void PartialMemsetEliminationTest1()
{
    MyStruct s;
    memset((ULONG_PTR)&s+4, 0, 16);

    s.Field1 = 1;
    // No init for fields 2 - 5
    s.Field6 = 6;

    TakeAddressOf(&s);
}
```

Yes! The compiler can trim the memset

Memset is adjusted to only initialize center fields

But should we always do this?

```
void PartialMemsetEliminationTest1()
{
    MyStruct s;
    memset(&s, 0, 24);

    s.Field3 = 1;

    TakeAddressOf(&s);
}
```

Field in the center of the structure is set

```
void PartialMemsetEliminationTest1()
{
    MyStruct s;
    memset(&s, 0, 8);
    memset((ULONG_PTR)&s+12, 0, 12);

    s.Field3 = 1;

    TakeAddressOf(&s);
}
```

Calling memset twice to avoid re-setting 4 bytes

This is extremely inefficient

Memset's are only trimmed/split if the compiler determines it is profitable

Small Structure Weirdness

```
void Foo()
{
    SmallStruct s1;
    memset(&s1, 0, 8);
    SmallStruct s2 = s1;
    baz(&s2);
}
```

Some functions that heavily use small structs have ~500 byte larger stack frame

Structs now allocated on the stack, get written to but never read

Small Structure Weirdness

Initializing structures results in a memset (that can later be unrolled)

```
void Foo()
{
    SmallStruct s1;
    memset(&s1, 0, 8);
    SmallStruct s2 = s1;
    baz(&s2);
}
```

Memset sticks around after machine-independent optimzations

Memset cannot be optimized as well as scalars, it hurts optimizations

Small memset later unrolled (scalar stores) but the damage is done

Scalar Replacement of Aggregates (SROA)

Convert scalar-size structure assignments to scalar stores EARLY

Can be optimized much better by all machine independent optimizations

```
void AfterSROAExample()
{
    SmallStruct s1;
    *(long long *)&s1 = 0;
    SmallStruct s2 = s1;
    baz(&s2);
}
```

Memset converted to scalar assignment early

This will also be a scalar assignment

Sometimes the compiler “unrolls” a memset into scalar stores

Only possible for small, fixed size memsets

Source Code	Not Optimized	Unrolled (Fastest)

`Memset(Foo, 0, 32);`	`mov rcx, &Foo`
	
	`xor rdx, rdx`
	`lea r8d, [rdx + 32]`
	
	`call memset`
		`mov rcx, &Foo`
		
		`xorps xmm0, xmm0`
		`movdqu [rcx], xmm0`
		`movdqu [rcx+16], xmm0`

Unrolled memsets can be optimized with scalar optimizations

Scalar optimizations often work a bit better than struct optimizations

Executes significantly (~4x) faster than calling memset

MSVC now unrolls memsets using XMM register stores

Saves code size compared to using normal scalar registers

Bigger memsets can be unrolled for minimal code size cost

Results

Final Performance Numbers

All performance targets met: Under 1% regression for real-world scenarios, under 2% regression for synthetic scenarios

Code size increase of around 0.1% for NT kernel

Only a single structure was opted out of InitAll in all Microsoft kernel-mode code

Optimizations also improved performance for video game engines (that aren’t using InitAll)

What We Shipped (Windows 10 1903)

Windows 1903 released to customers May 2019

InitAll for POD structures enabled for all Windows kernel-mode code, all Hyper-V code

4 vulnerabilities reported to Microsoft between April 2019 & July 2019 were killed by InitAll

Most researchers look at latest version, we expect reports to decrease for mitigated bugs

Developer Experience

InitAll is not enabled for debug builds or no-optimization builds

Hopefully this prevents developers from depending on it (for now)

Static analysis ignores InitAll initializations

Developers still receive warnings if they forget to initialize things

Compat Issues

Anti-cheat engines do some crazy things

Scan kernel code pages looking for specific byte patterns

Use this to locate undocumented functions

InitAll added initializations to some of these functions

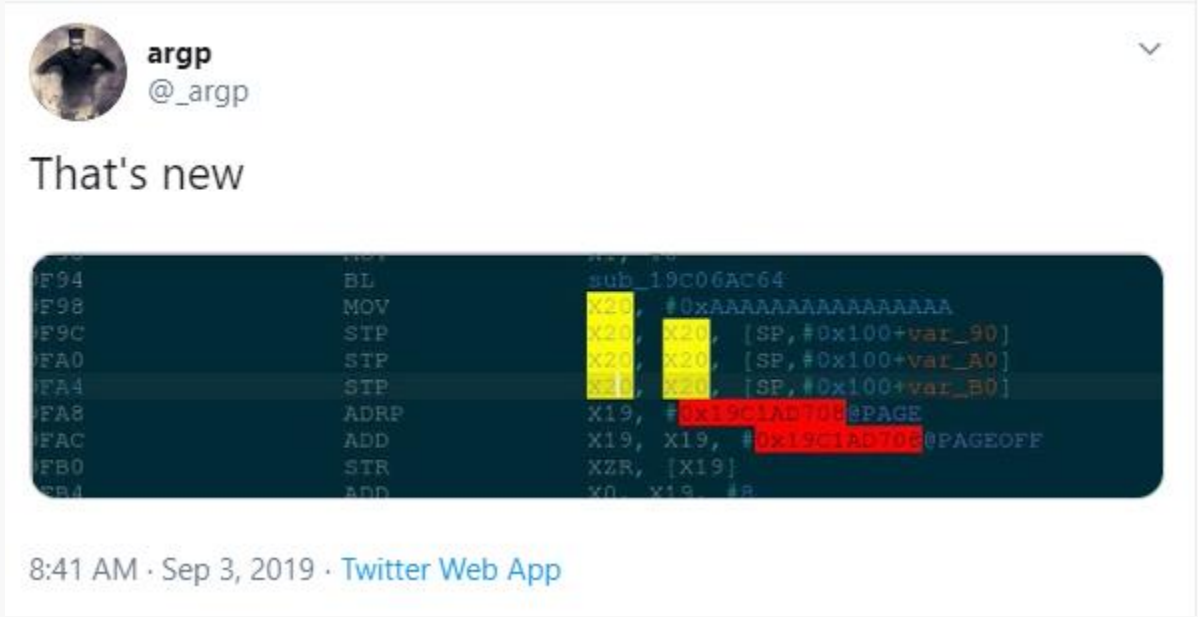
Anti-cheat couldn’t find the functions, kernel crashes ☹

Related Work

Auto initialization work happening in LLVM & GCC

Apparently Apple is now using this for iOS

Linus has expressed interest for the Linux kernel



iOS assembly showing new auto-initializations

What's Next?

Expanding what we apply InitAll to

Scalars, arrays of pointers

More code bases (not just kernel and Hyper-V)

Eventually: Everything on the stack (more testing needed)

New kernel heap allocators zero-by-default

Closing Thoughts

Uninitialized memory issues are unintuitive, hard to debug, easy to program, and lead to real world attacks against end users

Mostly unique to the C/C++ language

All major platforms (Windows, iOS, Linux) are moving towards auto-initialization

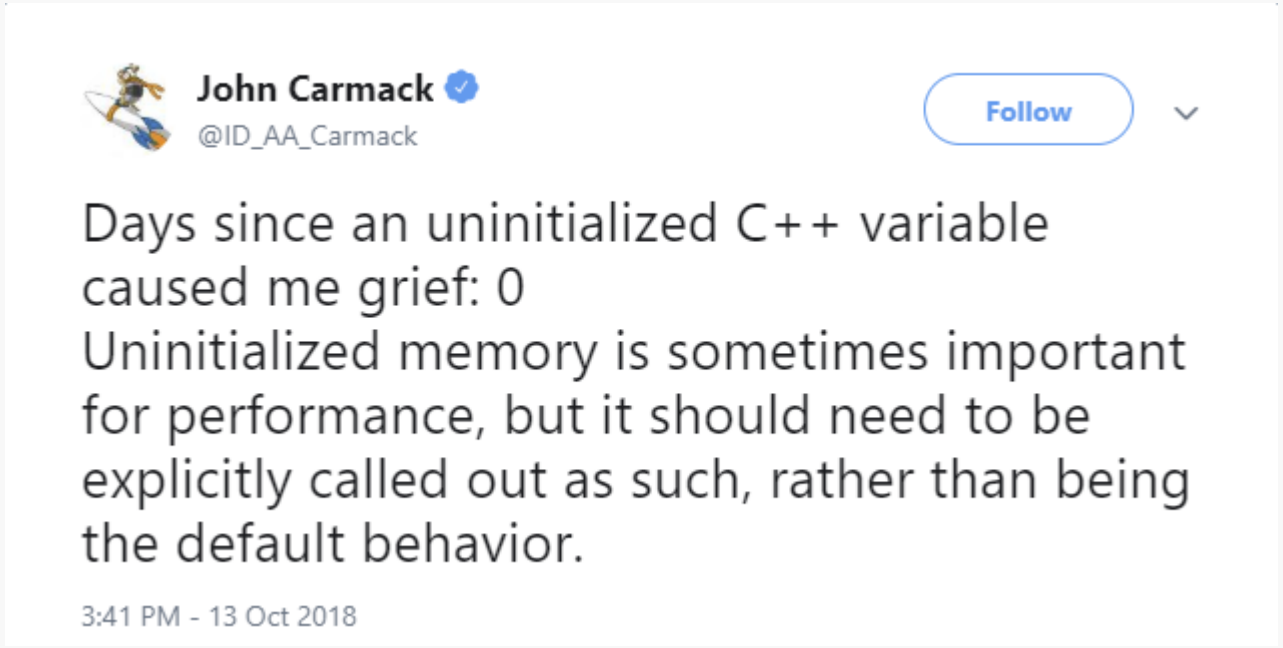
Closing Thoughts

The C & C++ language committee should do something about this problem

Why not initialize variables by default? Give users an “uninitialized” keyword to opt-out performance sensitive variables

It’s not just security people that hate uninitialized memory

Init-by-default wouldn’t be a breaking change in the language



Come say hi

Visit our Microsoft booth

Mon – Thu
Outside **Aurora D**

Help us learn

Take our survey https://aka.ms/cppcon

And have a chance to win an Xbox One S

Fri 9/20 16:15 – 18:00
De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable (“Simplifying C++” #6 of N)
Herb Sutter @ Aurora A

Other talks remaining from Microsoft

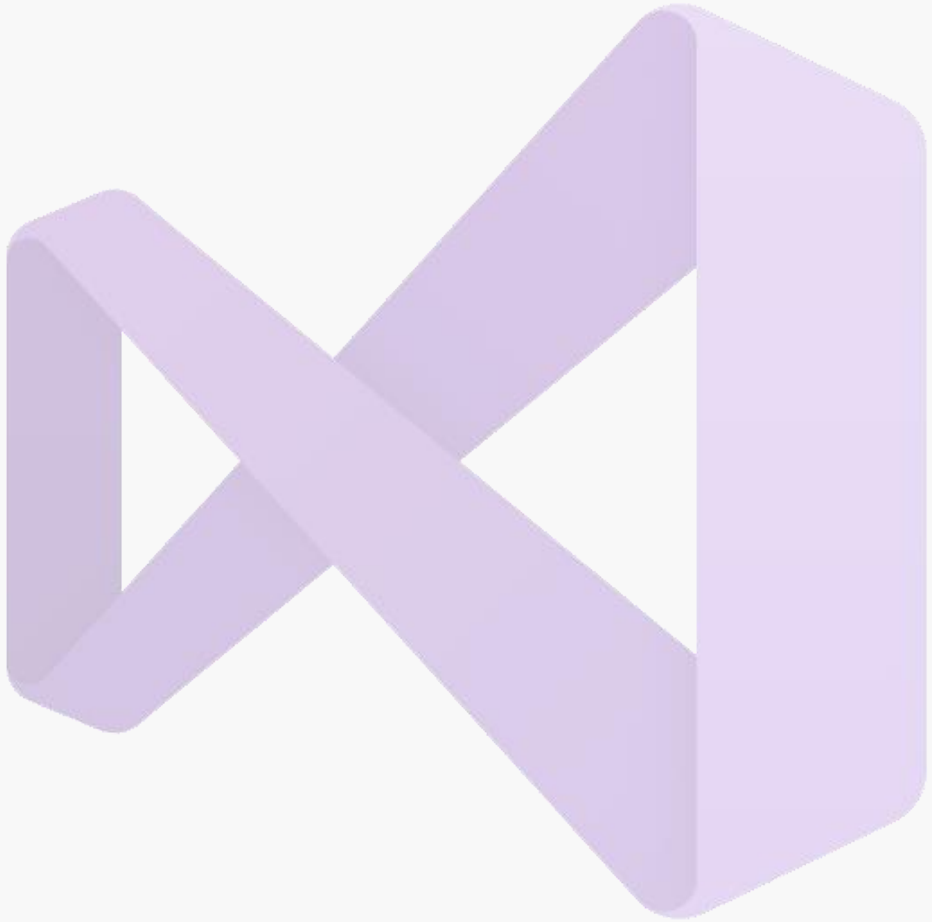
Thursday, September 19th

15:15 – 15:45: Don't Package Your Libraries, Write Packagable Libraries! (Part 2) by Robert Schumacher

16:45 – 17:45: Floating-Point charconv: Making Your Code 10x Faster With C++17's Final Boss by Stephan T. Lavavej

Friday, September 20th

16:15 – 18:00: De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable ("Simplifying C++" #6 of N) by Herb Sutter



References

References

- http://j00ru.vexillium.org/papers/2018/bochspwn_reloaded.pdf
- http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Flake.pdf
- http://www.ruxcon.org.au/files/2008/Uninitialized%20Variables%20-%20Live.ppt