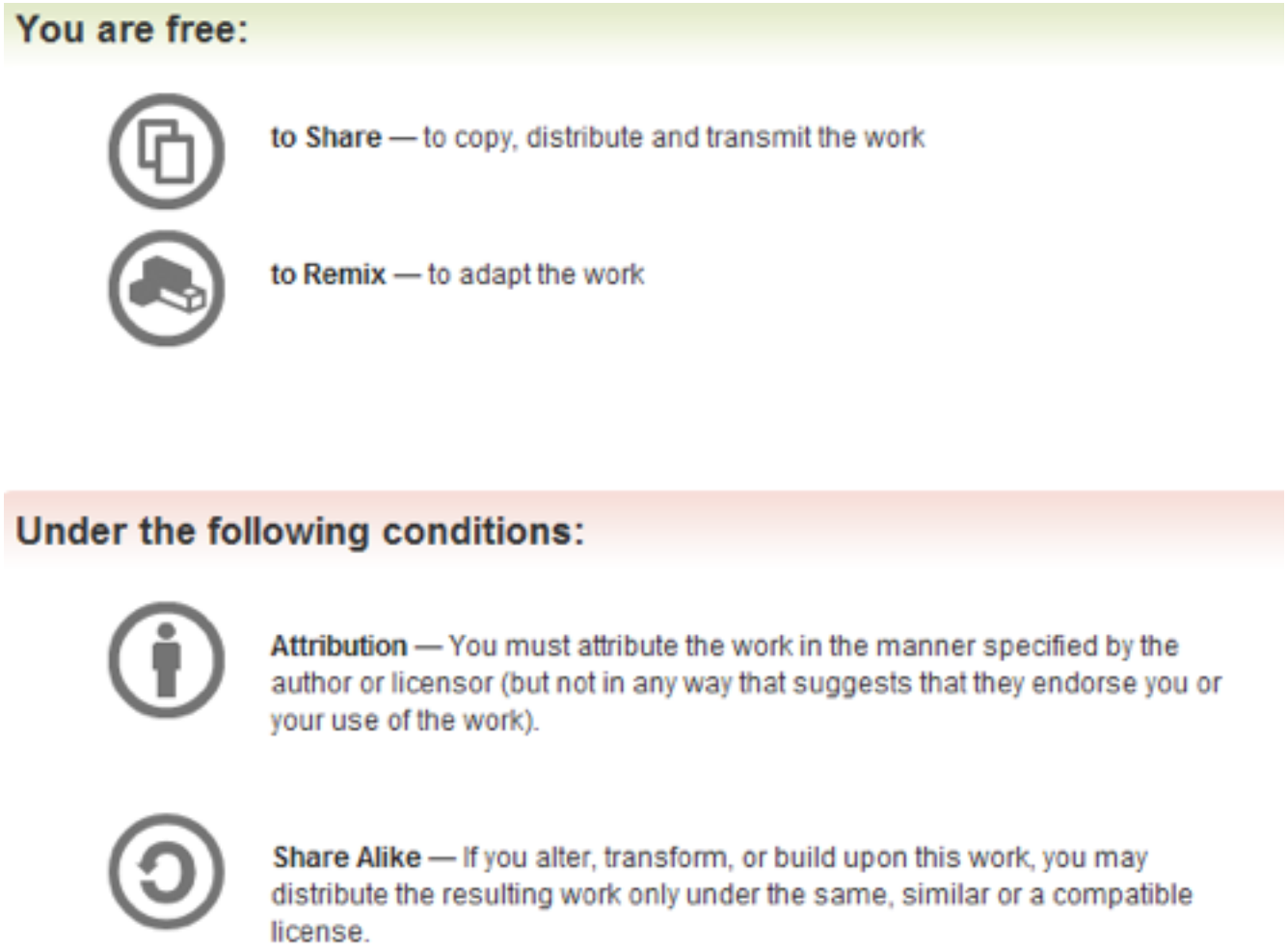
Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2014-2015

xeno@legbacore.com

All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



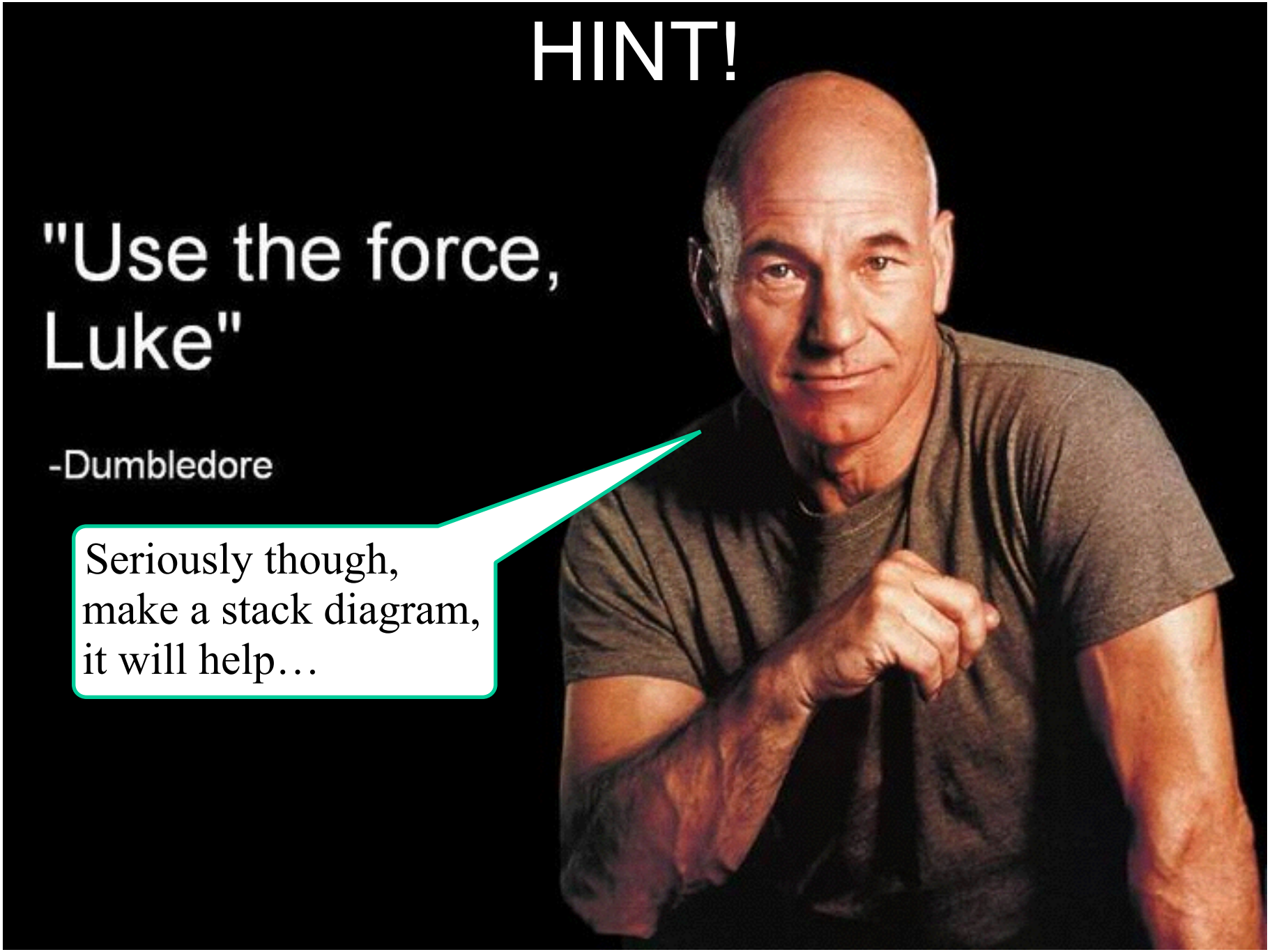
Attribution condition: You must indicate that derivative work
"Is derived from Xeno Kovah's 'Intro x86-64’ class, available at http://OpenSecurityTraining.info/IntroX86-64.html”

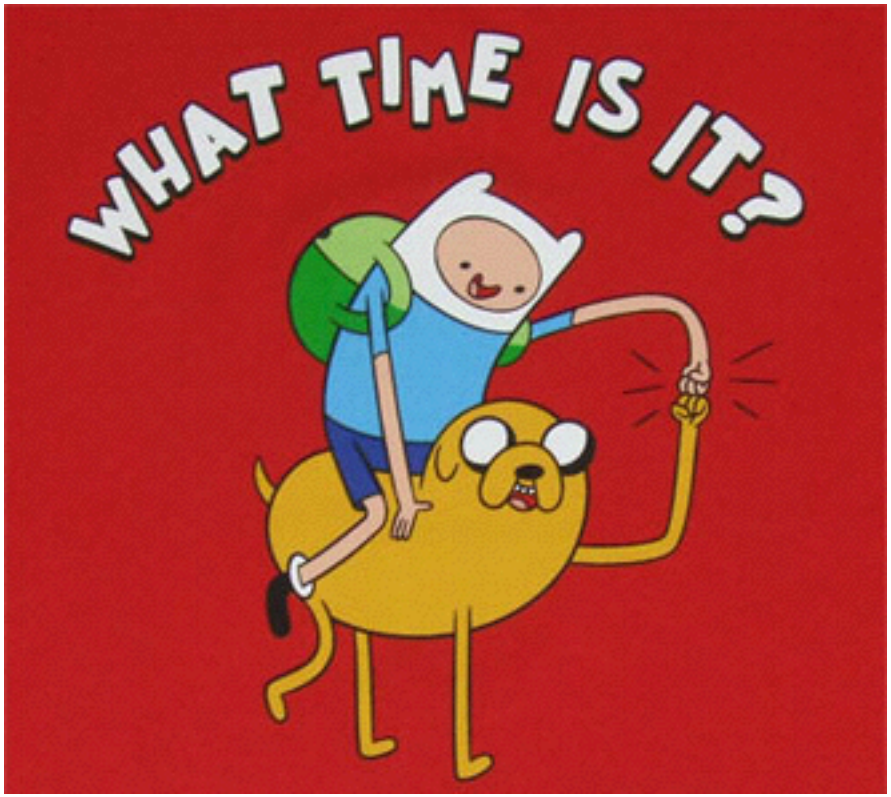
Are you ready to feel the POWER (buffer over)flow through you?!?!

BasicBufferOverflow.c: Don't be lame! Force it to call AwesomeSauce()!

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

int lame(__int64 value){
	__int64 array1[4];
	__int64 array2[4];
	array2[0] = array2[1] = array2[2] = array2[3] = value;
	memcpy(&array1, &array2, 4 * sizeof(__int64));
	return 1;
}

int lame2(__int64 size, __int64 value){
	__int64 array1[4];
	__int64 array2[4];
	array2[0] = array2[1] = array2[2] = array2[3] = value;
	memcpy(&array1, &array2, size * sizeof(__int64));
	return 1;
}

void AwesomeSauce(){
	printf("Awwwwwww yeaaahhhhh! All awesome, all the time!\n");
}

int main(unsigned int argc, char ** argv){
	__int64 size, value;
	size = _strtoi64(argv[1], "", 10);
	value = _strtoi64(argv[2], "", 16);

	if(!lame(value) || !lame2(size,value)){
		AwesomeSauce();
	}
	else{
		printf("I am soooo lame :(\n");
	}

	return 0xdeadbeef;
}
```

HINT!


"Use the force, Luke"
-Dumbledore
Seriously though, make a stack diagram, it will help…



Lab time!

Solution

Command Arguments	6 0x0000000140001110

Simple, right? :)

The stack looks like this at line 9 in lame() before the memcpy():

00000000`0012FE80	arg1 = ecx = value
00000000`0012FE78	return address = 0000000140010f0
00000000`0012FE70	undef
00000000`0012FE68	array2[3] = undef
00000000`0012FE60	array2[2] = undef
00000000`0012FE58	array2[1] = undef
00000000`0012FE50	array2[0] = undef
00000000`0012FE48	array1[3] = value
00000000`0012FE40	array1[2] = value
00000000`0012FE38	array1[1] = value
RSP+0x20 → 00000000`0012FE30	array1[0] = value

The stack looks like this at line 10 in lame() after the memcpy():

00000000`0012FE80	arg1 = ecx = value
00000000`0012FE78	return address = 00000001400010f0
00000000`0012FE70	undef
00000000`0012FE68	array2[3] = value
00000000`0012FE60	array2[2] = value
00000000`0012FE58	array2[1] = value
00000000`0012FE50	array2[0] = value
00000000`0012FE48	array1[3] = value
00000000`0012FE40	array1[2] = value
00000000`0012FE38	array1[1] = value
RSP+0x20 → 00000000`0012FE30	array1[0] = value

The stack looks like this at line 17 in lame2() before the memcpy(): addresses look familiar?

00000000`0012FE80	arg1 = ecx = value
00000000`0012FE78	return address = 0000000140001 0f0
00000000`0012FE70	undef
00000000`0012FE68	array2[3] = “undef” but still actually = value
00000000`0012FE60	array2[2] = “undef” but still actually = value
00000000`0012FE58	array2[1] = “undef” but still actually = value
00000000`0012FE50	array2[0] = “undef” but still actually = value
00000000`0012FE48	array1[3] = value
00000000`0012FE40	array1[2] = value
00000000`0012FE38	array1[1] = value
RSP+0x20 → 00000000`0012FE30	array1[0] = value

The stack looks like this at line 17 in lame2() before the memcpy(): addresses look familiar?

00000000`0012FE80	arg1 = ecx = value
00000000`0012FE78	return address = ***value! WINNER!***
00000000`0012FE70	value
00000000`0012FE68	array1[3] = value
00000000`0012FE60	array1[2] = value
00000000`0012FE58	array1[1] = value
00000000`0012FE50	array1[0] = value
00000000`0012FE48	array1[3] = value
00000000`0012FE40	array1[2] = value
00000000`0012FE38	array1[1] = value
00000000`0012FE30	array1[0] = value

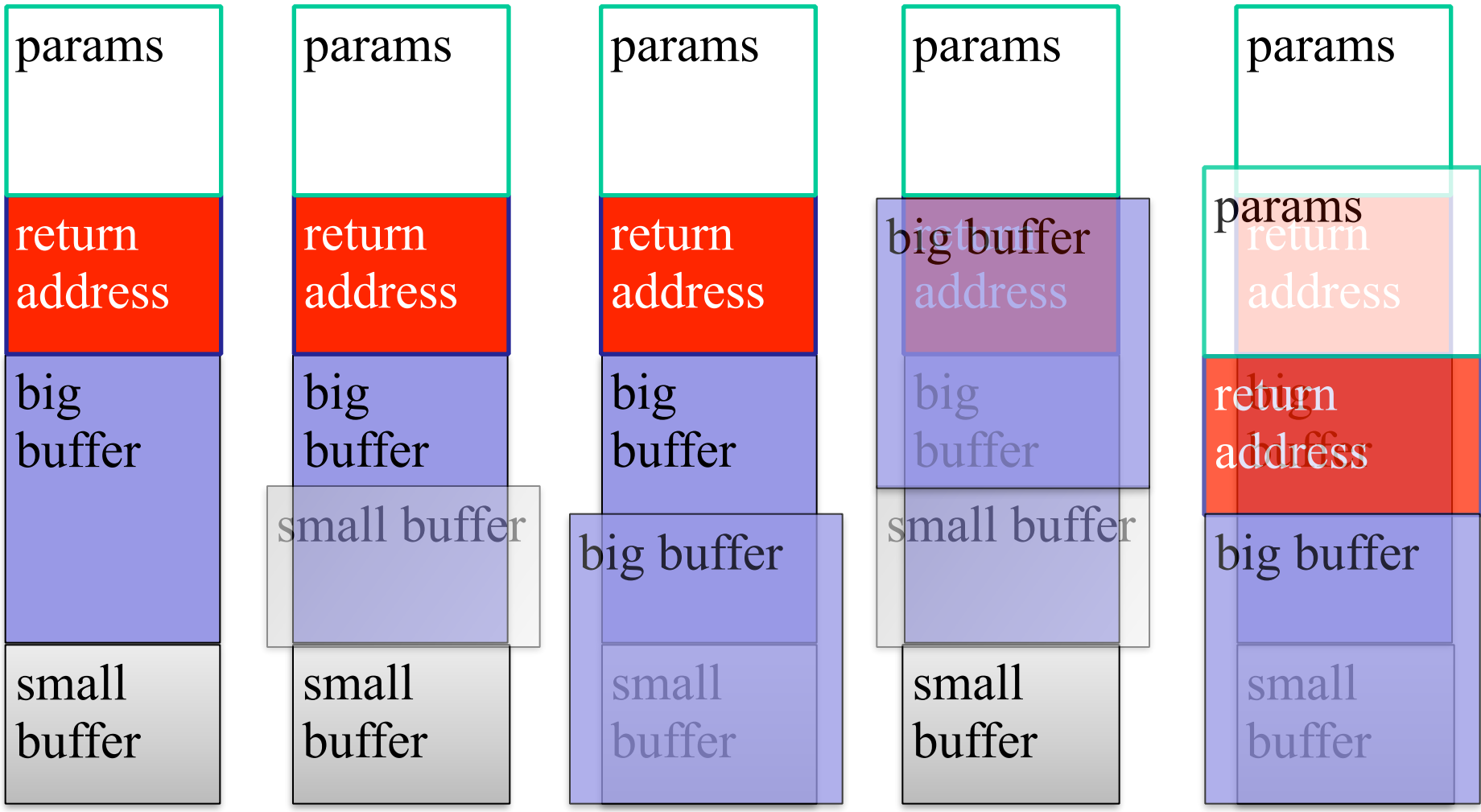
RSP+0x20 →

So set the size to 6 QWORDs to copy, and set the value to the address of AwesomeSauce() and collect your winner winner chicken dinner!

Exploit mitigations

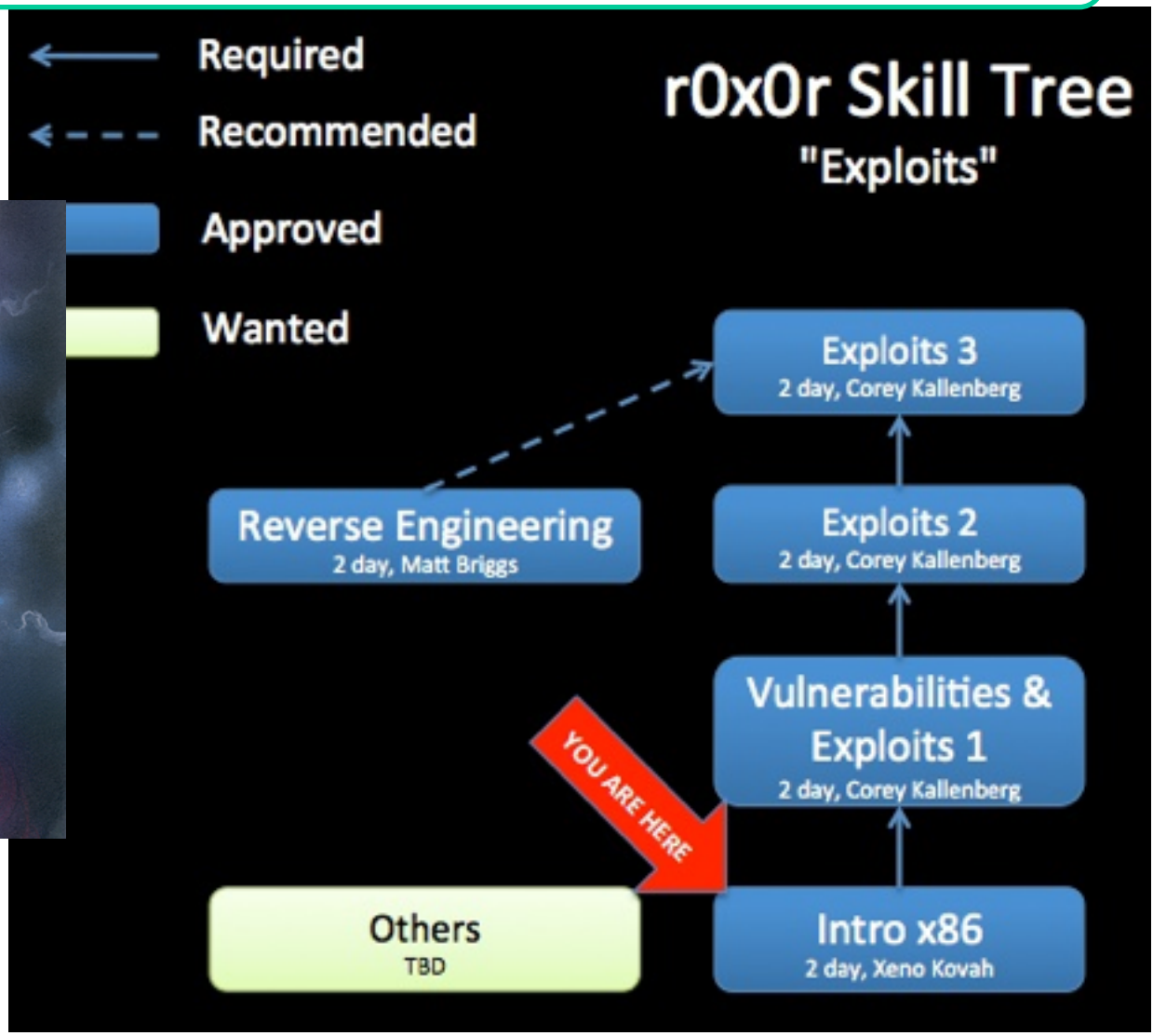
- This is a good example of how MS's compiler has exploit mitigations baked in.
- I was trying to just take my old, simple, and explicitly exploitable piece of code and port it to x86-64.
- But when I would put my buffers so that the big one would overwrite the small one, it placed the big buffer next to the stack pointer.
- This is smart since presumably writes will tend to be from the small to the big. But even if it's from the big to the small, it still won't naturally overflow into the stack pointer with contents from the buffer

Exploit mitigations 2



http://msdn.microsoft.com/en-us/library/bb430720.aspx

True power can not be had so simply, novice…
NX, ASLR, SEHOP…all these would stand in your way.
For a taste of true power, you must start start down a different path…a darker path



http://opensecuritytraining.info/Exploits1.html