

[www.linuxboot.org](http://www.linuxboot.org)

# LinuxBoot progress: boot anything from Linux

Chris Koch ([@hugelgupf](https://twitter.com/hugelgupf))

Ofir Weisse

Google

UMich

Platform Security Summit - October 1, 2019

with

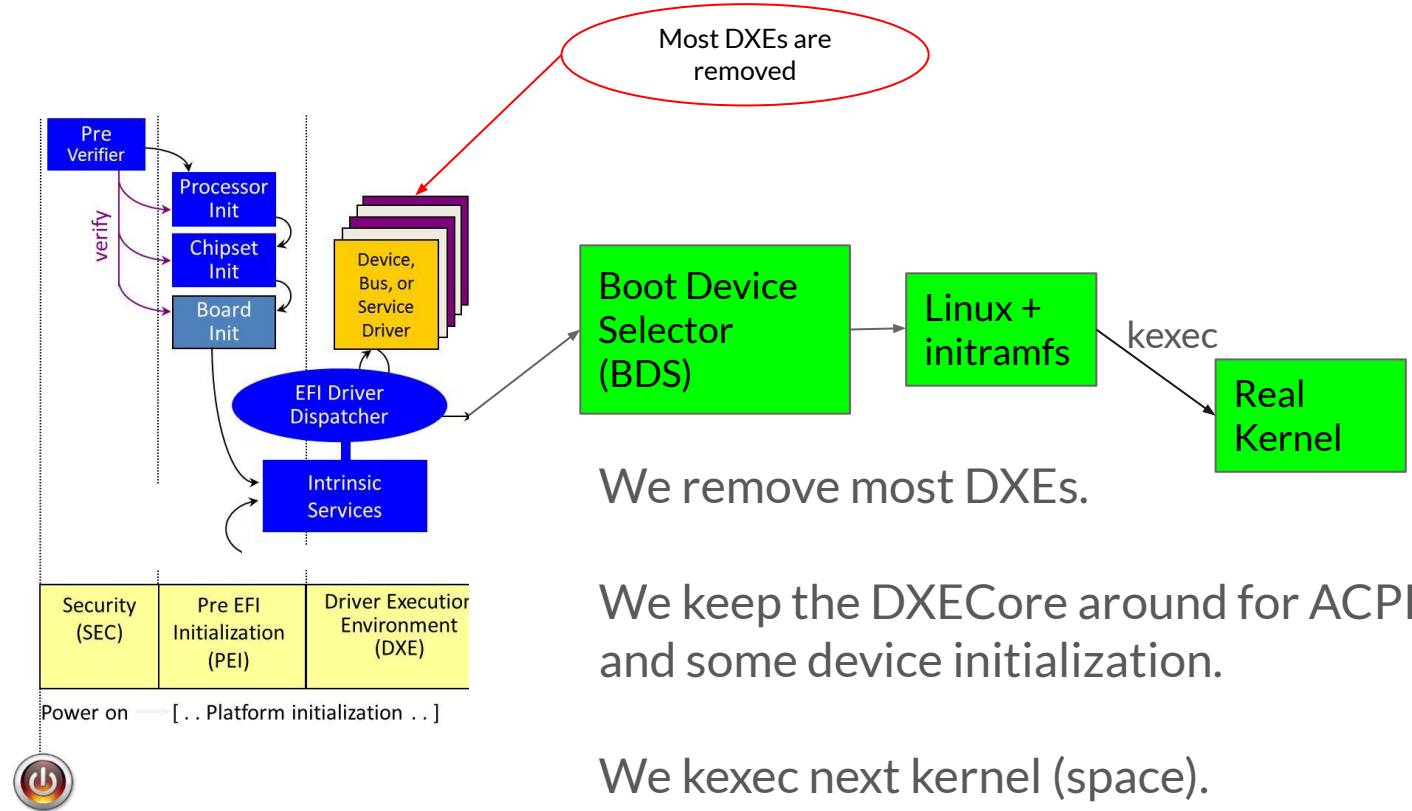
Ron Minnich, Ryan O'Leary, Gan Shun Lim, Max Shegai, Trammell Hudson,  
Jean-Marie Verdun, David Hendricks, Andrea Barberio, Philipp Deppenwiese **and many others**

[@hugelgupf](https://twitter.com/hugelgupf)

# Recap: LinuxBoot on UEFI

Linux knows how to initialize devices.

Compile kernel as a PE32 executable: EFI\_STUB



We remove most DXEs.

We keep the DXECore around for ACPI and some device initialization.

We kexec next kernel (space).

# Why?

We've replaced UEFI complexity with Linux Kernel complexity

Little review & visibility

VS

an uncountable number of contributors

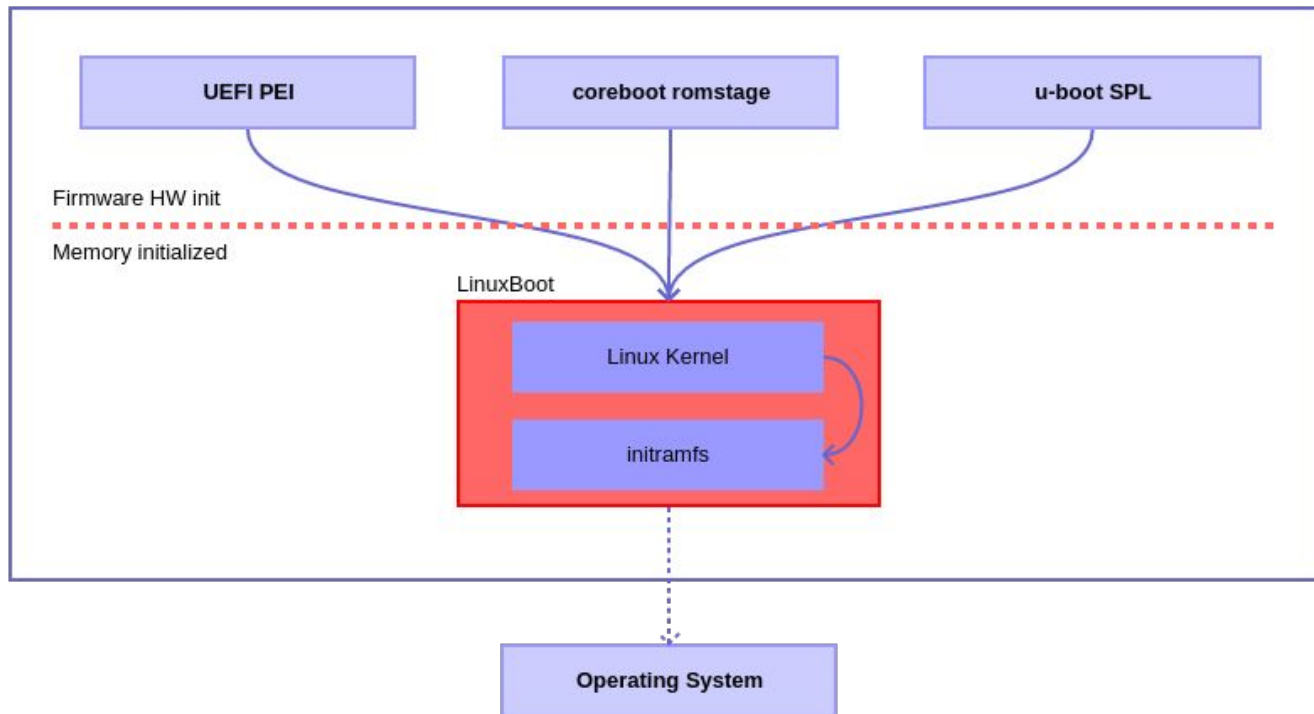
(Also, reproducible builds)

The screenshot displays three GitHub repository pages. Each page shows the repository name, navigation tabs (Code, Pull requests, Projects, Insights), and a summary bar with statistics. The 'contributors' count is highlighted with a red box in each case.

Repository	Commits	Branches	Releases	Contributors	License
torvalds / linux	723,054	1	537	∞	GPL-2.0
EDK II	22,924	8	1	144	
GRand Unified Bootloader	9,611	2	8	103	GPL-3.0

# Not on UEFI?

SPI Flash





# u-root: pointlessly rewriting things in Go

- We have the full toolset of Linux applications at our fingertips in firmware now.
  - Let's use them!
  - Let's use a memory-safe language.
  - Let's use a language that makes concurrency easy.
- Enter **u-root**: 3MB - 5MB (compressed) initramfs in **Go**
  - busybox-like tools (cat, dd, ls, cp, cpio, ...)
  - kexec-based bootloaders (PXE- and GRUB-compatible boot tools, ...)
  - [github.com/u-root/u-root](https://github.com/u-root/u-root)



# Progress: Project Health

- New: merging duplicate projects
  - systemboot + u-root: Facebook + Google
  - systemboot had TPM + certificate support
  - u-root the “cleaner” bootloaders
  - deduplicate + improve code health

## unit test in pkg/gpio



```
func TestExport(t *testing.T) {
    testutil.SkipIfNotRoot(t)

    if err := Export(10); err != nil {
        t.Errorf("Could not export pin 10: %v", err)
    }

    // Only 10-20 are valid GPIOs in the mock chip.
    if err := Export(9); err == nil {
        t.Errorf("Export(pin 9) should have failed, got nil")
    }
}
```

CI presubmit:

VM-based  
integration tests

(still needs work!)

## Launch VM with unit tests in pkg/gpio



```
func TestIntegration(t *testing.T) {
    vmtest.GolangTest(t, []string{"github.com/u-root/u-root/pkg/gpio"}, &vmtest.Options{
        QEMUOpts: qemu.Options{
            // Make GPIOs nums 10 to 20 available through the
            // mockup driver.
            KernelArgs: "gpio-mockup.gpio_mockup_ranges=10,20",
        },
    })
}
```

```
q, cleanup := vmtest.QEMUTest(t, &vmtest.Options{
```

```
    BuildOpts: uroot.Opts{
```

```
        Commands: uroot.BusyBoxCmds(
```

```
            "github.com/u-root/u-root/cmds/core/init",
```

```
            "github.com/u-root/u-root/cmds/core/kexec",
```

```
        ),
```

```
        ExtraFiles: []string{
```

```
            "/tmp/multibootKernel:kernel",
```

```
        },
```

```
    },
```

```
    Uinit: []string{
```

```
        `kexec -l kernel -e -d --module="/kernel foo=bar" --module="/sbin/bb"`,
```

```
    },
```

```
    QEMUOpts: qemu.Options{
```

```
        SerialOutput: &serial,
```

```
    },
```

```
})
```

```
defer cleanup()
```

```
if err := q.Expect(`"status": "ok"`); err != nil {
```

```
    t.Logf(string(serial.Bytes()))
```

```
    t.Fatalf(`expected '"status": "ok"', got error: %v`, err)
```

```
}
```

**Commands to build into VM initramfs**

**File to include in VM initramfs at /kernel**

**Command to run inside VM**

**Expect script**





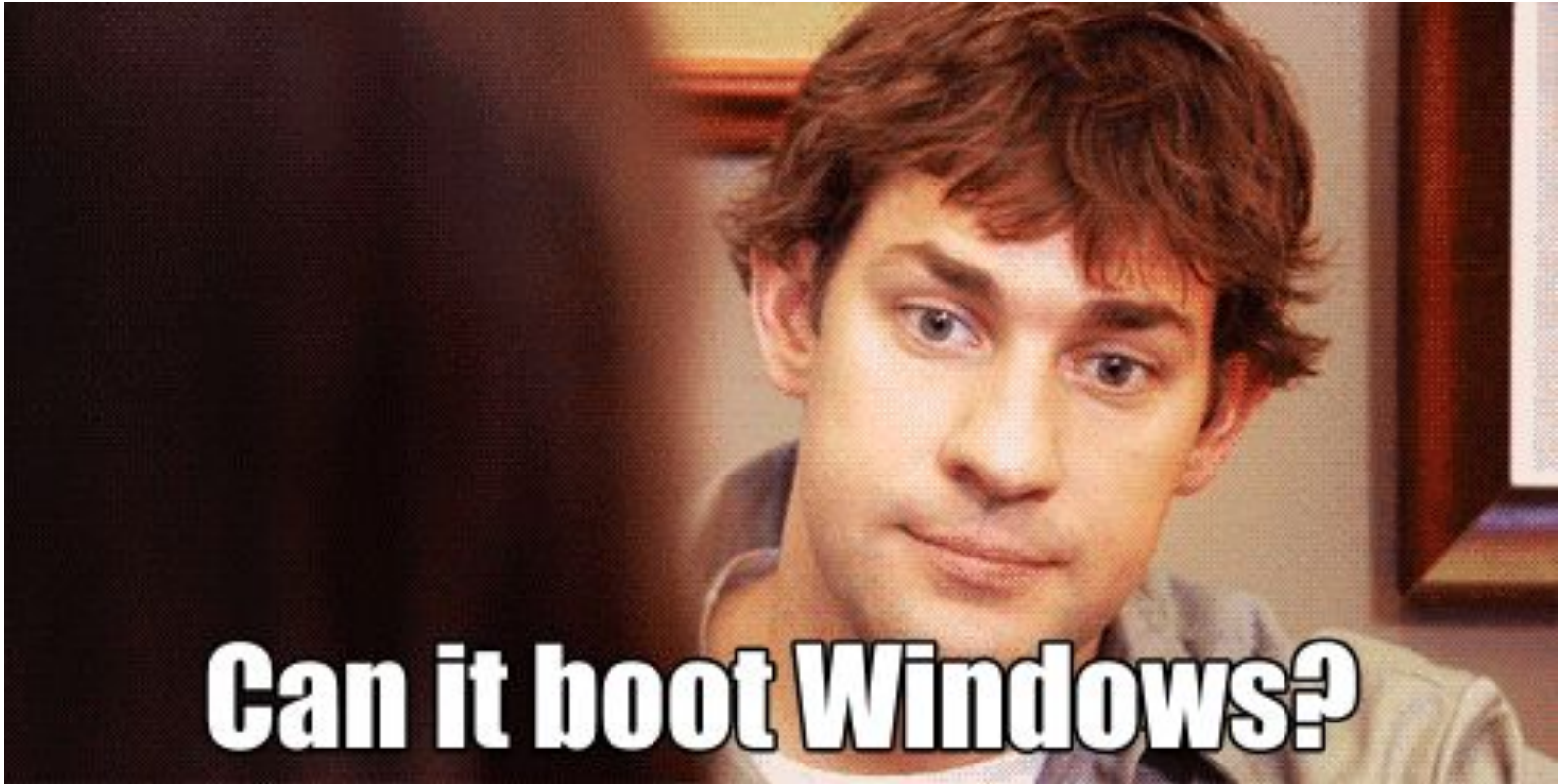
# Progress

- New: **kexec multiboot kernels (Xen, ESXi, tboot, ...)**
  - tboot support essential to some of our users
  - Imagine writing a trampoline in Go assembly :)

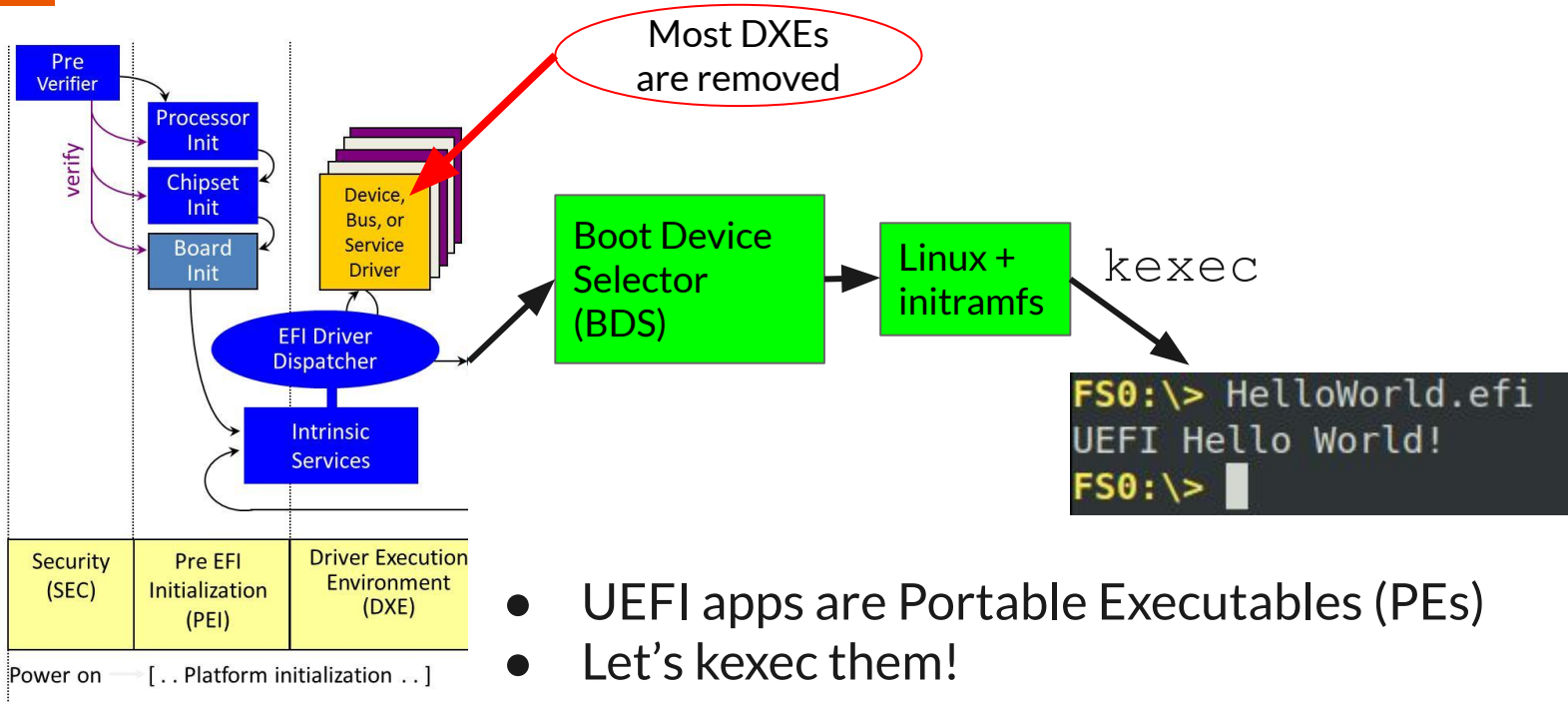
## func Load

```
func Load(debug bool, file, cmdline string, modules []string, ibft *ibft.IBFT) error
```

Load parses and loads a multiboot kernel `file` using `kexec_load`.



# Why not be UEFI compliant



- UEFI apps are Portable Executables (PEs)
- Let's kexec them!
- Specifically, kexec Windows boot-manager

# What do you need to boot EFI apps?

PE32 loader  
into ring 0

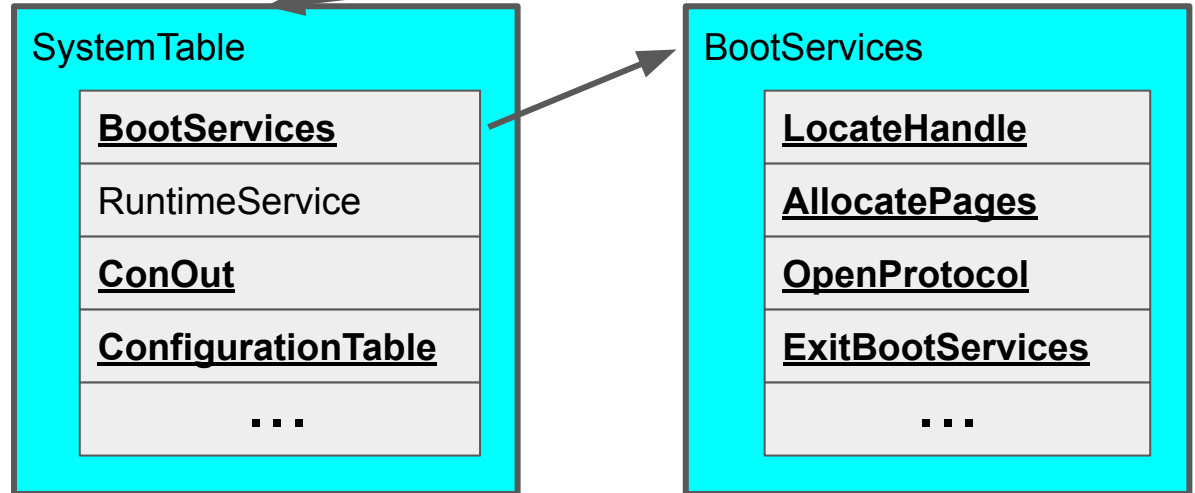
&

pointers to pass to  
the entrypoint

&

PE32 app expects  
physical addressing

```
ModuleEntryPoint ( EFI_HANDLE ImageHandle,  
                   EFI_SYSTEM_TABLE *SystemTable
```





## Let's collect some info

- Take EDKII, add a bunch of prints
  - What Boot Services are called?
  - What Protocols are invoked?
- Use that to boot Windows in a VM under EDKII/OVMF

# Minimum Required EFI Boot Services

---

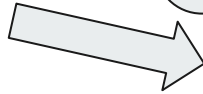
Services Used (out of 44)

- StartImage
- LocateHandle
- Open/CloseProtocol
- Allocate/FreePages
- Allocate/FreePool
- GetMemoryMap
- SetWatchdogTimer
- ExitBootServices

# Minimum Required EFI Protocols

## Services Used (out of 44)

- StartImage
- LocateHandle
- Open/CloseProtocol
- Allocate/FreePages
- Allocate/FreePool
- GetMemoryMap
- SetWatchdogTimer
- ExitBootServices



**Happy conclusion:**  
UEFI spec is huge, but only a small subset is actually used

## Protocols Used (Out of ~441)

- LoadedImageProtocol
- BlockIoProtocol
- DevicePathProtocol
- SimpleTextInputExProtocol
- StorageSecurityCommandProtocol
- GraphicsOutputProtocol



# Launching Windows via kexec

- 3 stage loading process
  - bootmgfw.efi
    - Loads winload.efi
      - which loads ntoskernel.exe
- kexec code for PE32 loader -- **not that hard**
  - validation for this is easy -- launch
    - Linux with PE32-shim
    - An EFI hello world





# Debug cycle



1. Launch



2. Crash



```
:0000000140566010 ; NTSTATU
:0000000140566010 public KI
:0000000140566010 KISystemS
:0000000140566010
:0000000140566010 var_ic8=
:0000000140566010 var_ic9=
:0000000140566010 var_b= qw
:0000000140566010
:0000000140566010 sub r
:0000000140566014 mov [
:0000000140566019 mov r
:000000014056601C mov c
:0000000140566023 mov e
:0000000140566028 mov r
:000000014056602F mov r
:0000000140566036 cmp d
:000000014056603B jnz snort_loc_140566042
```

3. Analyze



4. Learn something about 



5. Fix

# Debug cycle



# Debugging Example

```
0xffffffff80559d7d010    sub    $0x38,%rsp
0xffffffff80559d7d014    mov    %r15,0x30(%rsp)
0xffffffff80559d7d019    mov    %rsp,%r15
0xffffffff80559d7d01c    mov    %rcx,-0x22d03(%rip)
0xffffffff80559d7d023    mov    $0xffffffff,%ecx
0xffffffff80559d7d028    mov    -0x22d0f(%rip),%rdx
0xffffffff80559d7d02f    mov    0x88(%rdx),%r10
0xffffffff80559d7d036    cmpl  $0x0,0x24(%r10)
0xffffffff80559d7d03b    jne   0xffffffff80559d7d042
0xffffffff80559d7d03d    callq 0xffffffff8055a12e140
0xffffffff80559d7d042    mov    -0x22d29(%rip),%rcx
0xffffffff80559d7d049    mov    0x88(%rcx),%rdx
0xffffffff80559d7d050    mov    %rdx,%r10
```

```
Thread 1 hit Breakpoint 5, 0xffffffff80559d7d02f in ?? ()
(gdb) bt
#0  0xffffffff80559d7d02f in ?? ()
#1  0x0000000000000000 in ?? ()
(gdb)
```

- Illegal memory access
- Stack trace is useless

# Debugging Example

```
0xffffffff80559d7d010    sub    $0x38,%rsp
0xffffffff80559d7d014    mov    %r15,0x30(%rsp)
0xffffffff80559d7d019    mov    %rsp,%r15
0xffffffff80559d7d01c    mov    %rcx,-0x22d03(%rip)
0xffffffff80559d7d023    mov    $0xffffffff,%ecx
0xffffffff80559d7d028    mov    -0x22d0f(%rip),%rdx
0xffffffff80559d7d02f    mov    0x88(%rdx),%r10
0xffffffff80559d7d036    cmpl  $0x0,0x24(%r10)
0xffffffff80559d7d03b    jmp   0xffffffff80559d7d042
0xffffffff80559d7d03d    callq 0xffffffff8055a12e140
0xffffffff80559d7d042    mov    -0x22d29(%rip),%rcx
0xffffffff80559d7d049    mov    0x88(%rdx),%rdx
0xffffffff80559d7d050    mov    %rdx,%r10
```

```
Thread 1 hit Breakpoint 5, 0xffffffff80559d7d02f in ?? ()
(gdb) bt
#0  0xffffffff80559d7d02f in ?? ()
#1  0x00000000 in [224]: ntoskernel.addr2ImageAddr(0xffffffff80559d7d02f)
(gdb) Out[224]: (5374369839, '0x14056602f')
```

IPython:

```
Out[224]: (5374369839, '0x14056602f')
```

# Debugging Example

```
0xffffffff80559d7d010    sub    $0x38,%rsp
0xffffffff80559d7d014    mov    %r15,0x30(%rsp)
0xffffffff80559d7d019    mov    %rsp,%r15
0xffffffff80559d7d01c    mov    %rcx,-0x22d03(%rip)
0xffffffff80559d7d023    mov    $0xffffffff,%ecx
0xffffffff80559d7d028    mov    -0x22d0f(%rip),%rdi
0xffffffff80559d7d02f    mov    0x88(%rdx),%r10
0xffffffff80559d7d036    cmpl  $0x0,0x24(%r10)
0xffffffff80559d7d03b    jne   0xffffffff80559d7d042
0xffffffff80559d7d03d    callq 0xffffffff8055a12
0xffffffff80559d7d042    mov   -0x22d29(%rip),%rdi
0xffffffff80559d7d049    mov   0x88(%rcx),%rdi
0xffffffff80559d7d050    mov   %rdx,%r10
```

Jump to address

Jump address

Help Cancel OK

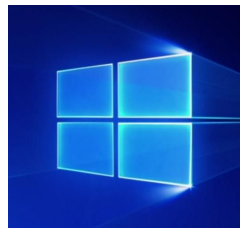
```
Thread 1 hit Breakpoint 5, 0xffffffff80559d7d02f
(gdb) bt
#0 0xffffffff80559d7d02f in ?? ()
#1 0x00000000 In [224]: ntoskernel.addr
(gdb) Out[224]: (5374369839, '0x14056602f')
```

```
0000000140566010 ; NTSTATUS __stdcall KiSystemStartup(PDRIVER_OBJECT DriverOb
0000000140566010 public KiSystemStartup
0000000140566010 KiSystemStartup proc near
0000000140566010
0000000140566010 var_1C8= qword ptr -1C8h
0000000140566010 var_1C0= qword ptr -1C0h
0000000140566010 var_8= qword ptr -8
0000000140566010
0000000140566010 sub    rsp, 38h
0000000140566014 mov    [rsp+38h+var_8], r15
0000000140566019 mov    r15, rsp
000000014056601C mov    cs:KeLoaderBlock_0, rcx
0000000140566023 mov    ecx, 0FFFFFFFFh
0000000140566028 mov    rdx, cs:KeLoaderBlock_0
000000014056602F mov    r10, [rdx+88h]
0000000140566036 cmp    dword ptr [r10+24h], 0
000000014056603B jnz   short loc_140566042
```



PAGELK:000000014056603D call KdInitSystem

# Challenge: Physical Addressing



1. AllocatePage

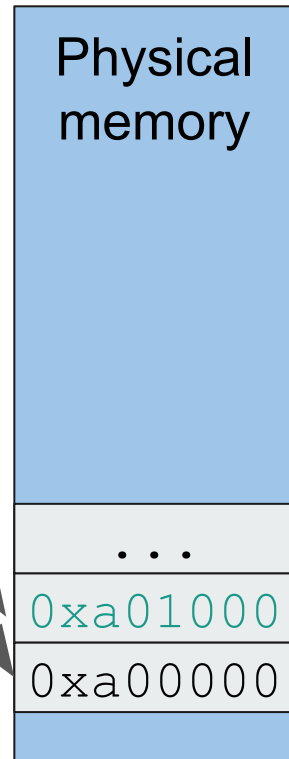
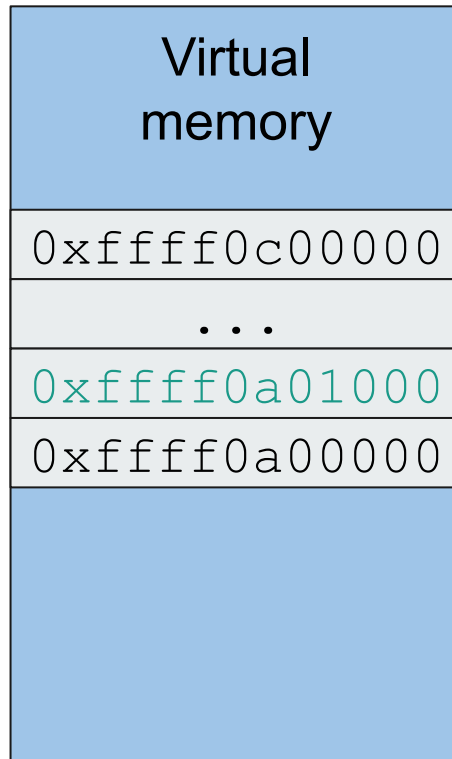


0xffff0a00000

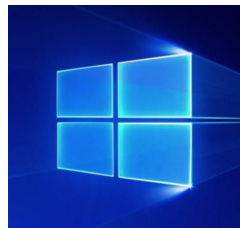


**LinuxBoot**

2. kmalloc



# Challenge: Physical Addressing



1. AllocatePage



0xffff0a00000

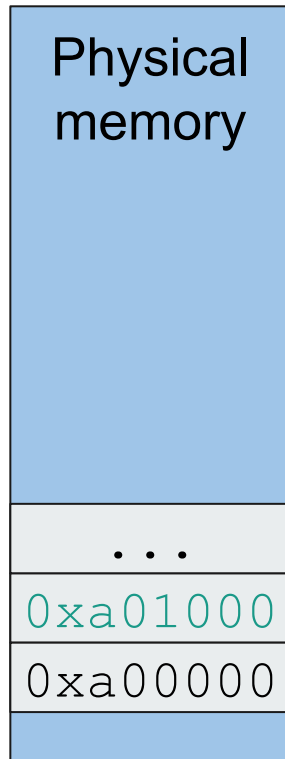


LinuxBoot

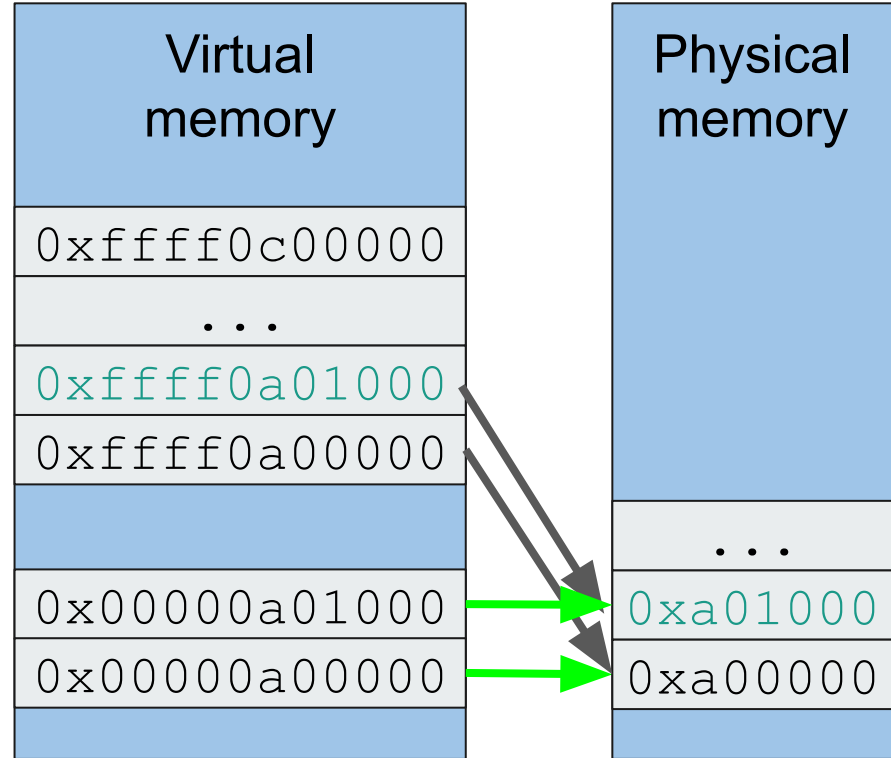
2. kmalloc

3. Change CR3

4. Access “physical” page 0xffff0a00000

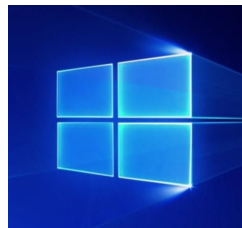


# Solution: 1:1 Virtual-to-Physical Mapping





# Solution: 1:1 Virtual-to-Physical Mapping



1. AllocatePage



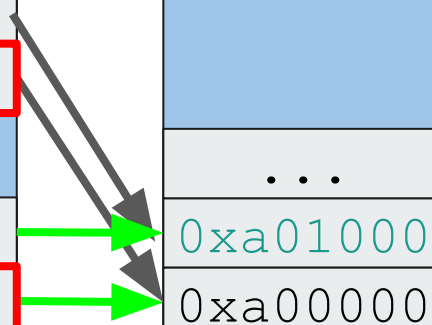
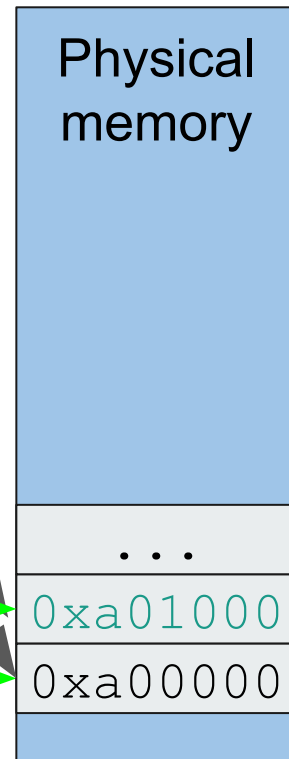
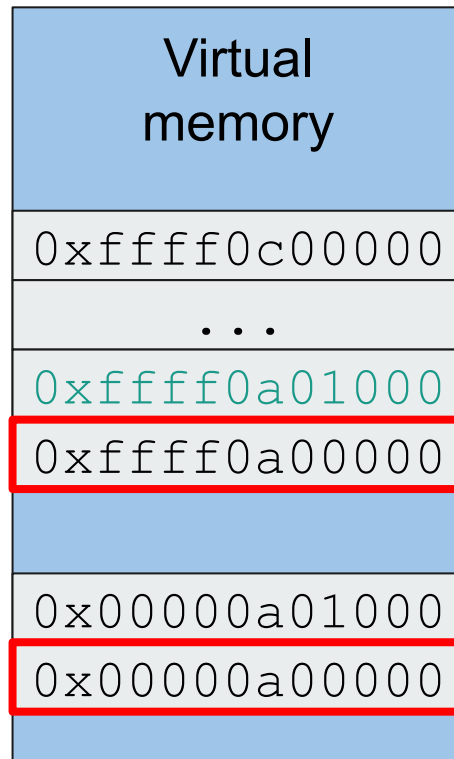
0xa00000



**LinuxBoot**

2. kmalloc

3. map 1:1



# Solution: 1:1 Virtual-to-Physical Mapping



1. AllocatePage



0xa00000



LinuxBoot

2. kmalloc

3. map 1:1

4. Change CR3

5. Access “physical” page 0xa00000



Physical  
memory

...

0xa01000

0xa00000

# Whoa moment 1

- ExitBootServices is called
- Most cores seems to be in KildleLoop
- 1 core in KiSystemCall64

```
(gdb) info threads
  Id Target Id          Frame
  1  Thread 1 (CPU#0 [halted ]) 0xffffffff8035aa94e0f in ?? ()
  2  Thread 2 (CPU#1 [halted ]) 0xffffffff8035aa94e0f in ?? ()
  3  Thread 3 (CPU#2 [halted ]) 0xffffffff8035aa94e0f in ?? ()
* 4  Thread 4 (CPU#3 [halted ]) 0xffffffff8035aa94e0f in ?? ()
  5  Thread 5 (CPU#4 [halted ]) 0xffffffff8035aa94e0f in ?? ()
  6  Thread 6 (CPU#5 [running]) 0xffffffff8035a665cf8 in ?? ()
```



chris @vegas/defcon  
@hugelgupf

Follow

Sneak peek at something our intern's been working on:

We're using Linux to implement EFI Boot Services to be able to kexec Windows from Linux.

We've reached the point where Windows calls ExitBootServices!!!

cc @mjb59 who blogged about this before

11:39 AM - 25 Jul 2019

43 Retweets 247 Likes





# Whoa moment 2

- Enabling kernel debug+EMS

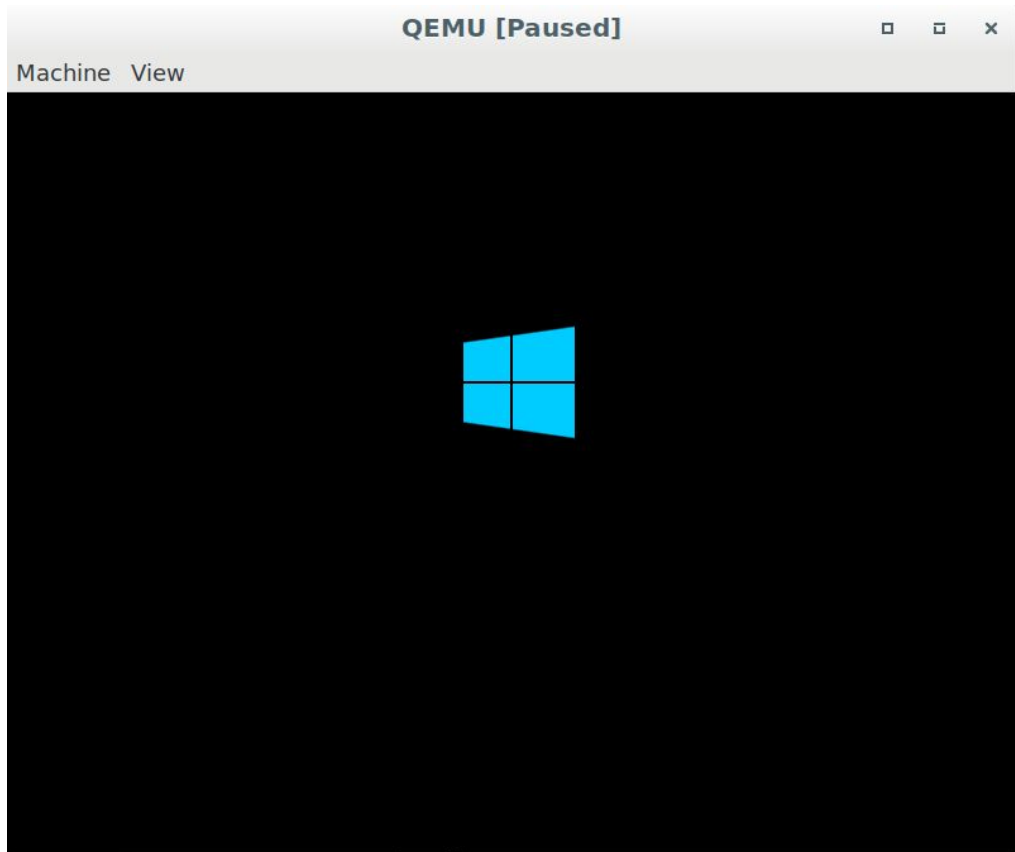
```
### efi_hook_GetMemoryMap:2461; 25: EfiConventionalMemory , 0x 104387000 -> 0x 0, 833, 0x000000000000000f
### efi_hook_GetMemoryMap:2461; 26: EfiLoaderCode , 0x 104800000 -> 0x 0, 7168, 0x000000000000000f
### efi_hook_GetMemoryMap:2461; 27: EfiConventionalMemory , 0x 106400000 -> 0x 0, 236544, 0x000000000000000f
### efi_hook_GetMemoryMap:2468; MemoryMapSize = 1344 MapKey = 0xee
### efi_hook_Exit()
0000000000000210 Z
" Z
Computer is booting, SAC started and initialized.
Use the "ch -?" command for information about using channels.
Use the "?" command for general help.
SAC>
EVENT: The CMD command is now available.
SAC>
```





## Whoa moment 3

- Enabling graphics
- Using Linux `/dev/fb0`
- Give the framebuffer to Windows
- And... we crash!





## Whoa moment 3

- Enabling graphics
- Using Linux `/dev/fb0`
- Step 1: Using it ourselves
- Step 2: Giving the framebuffer to Windows
- And... we crash!
- Fix..





## Whoa moment 3

- Enabling graphics
- Using Linux /dev/fb0
- Step 1: Using it ourselves
- Step 2: Giving the framebuffer to Windows
- And... we crash!
- Fix..
- Login!





Sign up for the Open Source Firmware Slack  
<http://slack.u-root.com>

## Code

[https://github.com/oweisse/u-root/tree/kexec\\_test/booting\\_windows](https://github.com/oweisse/u-root/tree/kexec_test/booting_windows)

<https://github.com/oweisse/linux>

## Thanks to

Ofir Weisse for being a productive PhD intern :)

## Me

[chrisko@google.com](mailto:chrisko@google.com)

<https://twitter.com/hugelgupf>