Replace your exploit-ridden firmware with a Linux kernel

Ron Minnich, Gan-shun Lim, Ryan O'Leary, Chris Koch, Xuan Chen

Google

Trammell Hudson Two Sigma
Andrey Mirtchovskii Cisco
Jean-Marie Verdun
Guillaume Giamarchi
Splitted-Desktop
Results

- OCP boot time: 8 minutes -> 17 seconds
  - i.e. 32x speedup
  - This is to a shell prompt in Linux
- OCP -> DHCP -> wget -> kexec: 20 seconds
- All userland written in Go
- Linux performance and reliability in firmware
- Eliminate all UEFI/ME post-boot activity
The problem

- Linux no longer controls the x86 platform
- Between Linux and the hardware are at least 2 ½ kernels
- They are completely proprietary and (perhaps not surprisingly) exploit-friendly
- And the exploits can persist, i.e. be written to FLASH, and you can’t fix that
## The operating systems

<table>
<thead>
<tr>
<th>Code you know about</th>
<th>Code you don’t know about</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring 0 (Linux)</td>
<td>Ring -2 kernel and ½ kernel</td>
</tr>
<tr>
<td>Ring -1 (Xen etc.)</td>
<td>Control all CPU resources.</td>
</tr>
<tr>
<td>Ring 3 (User)</td>
<td>Invisible to Ring -1, 0, 3</td>
</tr>
</tbody>
</table>

- **Ring -2 kernel and ½ kernel**
  - Control all CPU resources.
  - Invisible to Ring -1, 0, 3

- **SMM ½ kernel. Traps to 8086 16-bit mode.**

- **UEFI kernel running in 64-bit paged mode.**

- **Ring -3 kernels**
  - Management Engine, ISH, IE.
  - Higher privilege than Ring -2.
  - Can turn on node and reimage disks invisibly. Minix 3.

| X86 CPU you know about | X86 CPU(s) you don’t know about |
What’s in ring -2 and ring -3?

- IP stacks (4 and 6)
- File systems
- Drivers (disk, net, USB, mouse)
- Web servers
- Passwords (yours)
- Can reimage your workstation even if it’s powered off
Ring -3 OS: ME (Management Engine)

- Full Network manageability
- Regular Network manageability
- Manageability
- Small business technology
- Level III manageability
- IntelR Anti-Theft (AT)
- IntelR Capability Licensing Service (CLS)
- IntelR Power Sharing Technology (MPC)
- ICC Over Clocking
- Protected Audio Video Path (PAVP)
- IPV6
- KVM Remote Control (KVM)
- Outbreak Containment Heuristic (OCH)
- Virtual LAN (VLAN)
- TLS
- Wireless LAN (WLAN)
Vassilios Ververis: https://goo.gl/j7Jmx5

- Great overview of many early ME flaws
- Summary: just about every part of the ME software can be attacked
- Only some of the bugs get fixed ...
‘Intel ME exploit’: 50M hits

- “Wired” headline: “HACK BRIEF: INTEL FIXES A CRITICAL BUG THAT LINGERED FOR 7 DANG YEARS”
- How many is that? One billion systems?
- Bug was in the built-in web server in the ME
  - Yep: the hidden CPU had a web server
  - That evidently you can’t turn off
  - Even though docs said you could
Ring -2 “½ OS”: System Management Mode (SMM)

- Originally used for power management
- No time for full details but …
  - Vectors to 8086 16-bit mode code
    - I.e. great place for an attack
  - All kinds of interrupts can go here, e.g. USB
  - Nowadays almost all of these go out again to ACPI
- That said, it’s a very nasty bit of code
- Vendors use it as secret way to “value-add”
Are there SMI exploits?

- “system management interrupt exploit” -- 630K hits
- So, yes.
- Chipsets guarantee that once SMM is installed, can’t change it, see it, turn it off
  - SMM “hidden” memory at top 8 MiB of DRAM.
- SMM maintains vendor control over … you
Ring -2 OS: UEFI

- UEFI runs on the main CPU
- Extremely complex kernel
- Millions of lines of code
- UEFI applications are active after boot
- Security model is obscurity
Are there UEFI exploits?

- Absolutely
- Since UEFI (and only UEFI) can rewrite itself
  - These exploits can be made persistent
- You might even have UEFI fake the process of removing an exploit
- The only fix? A shredder
## (Some) UEFI components

<table>
<thead>
<tr>
<th>Component</th>
<th>Component</th>
<th>Component</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>CsmVideo</td>
<td>ArpDxe</td>
<td>Udp6Dxe</td>
<td>UsbKbDxe</td>
</tr>
<tr>
<td>Terminal</td>
<td>SnpDxe</td>
<td>IpSecDxe</td>
<td>UsbMouseDxe</td>
</tr>
<tr>
<td>SBAHCI</td>
<td>MnpDxe</td>
<td>UNDI</td>
<td>UsbBusDxe</td>
</tr>
<tr>
<td>AHCI</td>
<td>UefiPxeBcDxe</td>
<td>IsaBusDxe</td>
<td>XhciDxe</td>
</tr>
<tr>
<td>AhciSmm</td>
<td>NetworkStackSetupScreen</td>
<td>IsaIoDxe</td>
<td>USB/XHCI/etc</td>
</tr>
<tr>
<td>BIOSBLKIO</td>
<td>TcpDxe</td>
<td>IsaserialDxe</td>
<td>Legacy8259</td>
</tr>
<tr>
<td>IdeSecurity</td>
<td>Dhcp4Dxe</td>
<td>DiskIoDxe</td>
<td>DigitalTermometerSensor (sic)</td>
</tr>
<tr>
<td>IDESMM</td>
<td>Ip4ConfigDxe</td>
<td>ScsiBus</td>
<td></td>
</tr>
<tr>
<td>CSMCORE</td>
<td>Ip4Dxe</td>
<td>Scsidisk</td>
<td></td>
</tr>
<tr>
<td>HeciSMM</td>
<td>Mtftp4Dxe</td>
<td>GraphicsConsoleDxe</td>
<td></td>
</tr>
<tr>
<td>AINT13</td>
<td>Udp4Dxe</td>
<td>CgaClassDxe</td>
<td></td>
</tr>
<tr>
<td>HECIDXE</td>
<td>Dhcp6Dxe</td>
<td>SetupBrowser</td>
<td></td>
</tr>
<tr>
<td>AMITSE</td>
<td>Ip6Dxe</td>
<td>EhciDxe</td>
<td></td>
</tr>
<tr>
<td>DpcDxe</td>
<td>Mtftp6Dxe</td>
<td>UhciDxe</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

- 2 ½ hidden OSes in your Intel x86 system
- They have many capabilities
- They have network stacks and web servers
- They implement self-modifying code that can persist across power cycles and reinstalls
- They hide, have bugs, and control Linux
- Exploits have happened
- Scared yet? We sure are!
Can we fix this mess?

- Partially ...
  - Moving to AMD is not a solution, they’re closed too
  - Don’t believe all you read about Ryzen
- We focus on Intel x86 for now
- Reduce the scope of the 2 ½ OSes
- Overall project is called NERF
- Non-Extensible Reduced Firmware
  - Extensibility Considered Harmful
Non-Extensible Reduce Firmware

- Make firmware less capable of doing harm
- Make its actions more visible
- Remove all runtime components
  - Well, almost all: the ME is very hard to kill
  - But we took away its web server and IP stack
- Remove UEFI IP stack and other drivers
- Remove ME/UEFI self-reflash capability
- Linux manages flash updates
NERF components

- De-blobbed ME ROM
- UEFI ROM reduced to its most basic parts
- SMM disabled or vectored to Linux
- Linux kernel
- Userland written in Go (http://u-root.tk)
Removing the ME

- We don’t want ME at all; not an option
- If you remove ME firmware, your node
  - May never work again
  - May not power on (as in OCP nodes)
  - May power on, but will turn off in thirty minutes
- Good news: ME firmware has components
- And most are removable
  - Thanks Trammell Hudson
Removing most of the ME code

- me_cleaner can remove ME blobs
- [https://github.com/corna/me_cleaner](https://github.com/corna/me_cleaner)
- On minnowmax, 5M of 8M FLASH is ME
- me_cleaner.py reduces it to 300K
- Removes web server, IP stack, pretty much all the things you don’t want “Ring -3” doing
- Server (SPS) is not yet solved
## Me_cleaner on the minnowmax

<table>
<thead>
<tr>
<th>Component</th>
<th>Compression</th>
<th>Memory Range</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUP</td>
<td>Uncomp.</td>
<td>0x045000 - 0x05a000</td>
<td>NOT removed, essential</td>
</tr>
<tr>
<td>KERNEL</td>
<td>Uncomp.</td>
<td>0x05a000 - 0x08d000</td>
<td>removed</td>
</tr>
<tr>
<td>POLICY</td>
<td>Uncomp.</td>
<td>0x08d000 - 0x0a8000</td>
<td>removed</td>
</tr>
<tr>
<td>HOSTCOMM</td>
<td>Uncomp.</td>
<td>0x0a8000 - 0x0c0000</td>
<td>removed</td>
</tr>
<tr>
<td>FPF</td>
<td>Uncomp.</td>
<td>0x0c0000 - 0x0c6000</td>
<td>removed</td>
</tr>
<tr>
<td>RSA</td>
<td>LZMA</td>
<td>0x0c6000 - 0x0cc385</td>
<td>removed</td>
</tr>
<tr>
<td>fTPM</td>
<td>LZMA</td>
<td>0x0cd000 - 0x0dc305</td>
<td>removed</td>
</tr>
<tr>
<td>ClsPriv</td>
<td>Uncomp.</td>
<td>0x0dd000 - 0x0df000</td>
<td>removed</td>
</tr>
<tr>
<td>CLS</td>
<td>Uncomp.</td>
<td>0x0df000 - 0x0e8000</td>
<td>removed</td>
</tr>
<tr>
<td>SessMgr</td>
<td>LZMA</td>
<td>0x0e8000 - 0x0f3906</td>
<td>removed</td>
</tr>
<tr>
<td>TDT</td>
<td>LZMA</td>
<td>0x0f4000 - 0x0f9452</td>
<td>removed</td>
</tr>
</tbody>
</table>
It’s an eye test on OCP ...

<table>
<thead>
<tr>
<th>BUP</th>
<th>b2c2962872f9efb7fc9055c5a56c6e47565406eefe350de75ea52c4c3ef264 plain</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUP</td>
<td>1a24f589b04499cb7dcdbd48155294494660f484912738cf6ebcb9a1dbfe589f plain</td>
</tr>
<tr>
<td>KERNEL</td>
<td>5b419f959814a4dbda06fdcba4b84ed1a2488a2acb2de1ca2234807ba6d4fa [MATCH]</td>
</tr>
<tr>
<td>POLICY</td>
<td>c84a79ee14d7231bd8e967fc8660228bb4f5d75a6c516247d1435cf5d266f46f [MATCH]</td>
</tr>
<tr>
<td>HOSTCOMM</td>
<td>5e54d9f801aebc3957ff83ea7b6b34e5209e9ed14252457cbf751019932ea92f [MATCH]</td>
</tr>
<tr>
<td>ICCMOD</td>
<td>ee1a0bb4602da9c7e1669e85a54701c50ff33013ae4c40f8dcb25faddf82f8fb [MATCH]</td>
</tr>
<tr>
<td>BASEEXT</td>
<td>84074ba8ba4b6dca24e086be37d8c768468b63e18d1ac5909ba1f8e1d054f49b [MATCH]</td>
</tr>
<tr>
<td>SC</td>
<td>5b22b84a4ac67751a55c280ce6b69c2c9d6d649a9710031db43a14f39e4a337d [MATCH]</td>
</tr>
<tr>
<td>NM</td>
<td>ba2ff3a6803517408a50da2fffab21c6de16b84838c9ff7159ce3ec99e0c5261 [MATCH]</td>
</tr>
<tr>
<td>DM</td>
<td>91cbb5777bb5c5a3c2776edf15dc35b65b6eba2ae83d0b7ea03cb080e95ea83 [MATCH]</td>
</tr>
<tr>
<td>BUP</td>
<td>1a24f589b04499cb7dcdbd48155294494660f484912738cf6ebcb9a1dbfe589f plain</td>
</tr>
<tr>
<td>KERNEL</td>
<td>5b419f959814a4dbda06fdcba4b84ed1a2488a2acb2de1ca2234807ba6d4fa [MATCH]</td>
</tr>
<tr>
<td>POLICY</td>
<td>c84a79ee14d7231bd8e967fc8660228bb4f5d75a6c516247d1435cf5d266f46f [MATCH]</td>
</tr>
<tr>
<td>HOSTCOMM</td>
<td>5e54d9f801aebc3957ff83ea7b6b34e5209e9ed14252457cbf751019932ea92f [MATCH]</td>
</tr>
<tr>
<td>ICCMOD</td>
<td>ee1a0bb4602da9c7e1669e85a54701c50ff33013ae4c40f8dcb25faddf82f8fb [MATCH]</td>
</tr>
<tr>
<td>BASEEXT</td>
<td>84074ba8ba4b6dca24e086be37d8c768468b63e18d1ac5909ba1f8e1d054f49b [MATCH]</td>
</tr>
<tr>
<td>SC</td>
<td>5b22b84a4ac67751a55c280ce6b69c2c9d6d649a9710031db43a14f39e4a337d [MATCH]</td>
</tr>
<tr>
<td>NM</td>
<td>ba2ff3a6803517408a50da2fffab21c6de16b84838c9ff7159ce3ec99e0c5261 [MATCH]</td>
</tr>
<tr>
<td>DM</td>
<td>91cbb5777bb5c5a3c2776edf15dc35b65b6eba2ae83d0b7ea03cb080e95ea83 [MATCH]</td>
</tr>
</tbody>
</table>
Ring -2: Dealing with SMM

- We have experimental work that directs SMM interrupts to kernel handler
- Requires that kernel run before SMM is installed
- Or that SMM never be installed
- Most preferred: kill SMM
- Second: vector SMI# to kernel
There’s a huge amount of capability in UEFI
  ○ I.e. a great place to put exploits
Some interrupts still go there
  ○ SECDED
We want to remove those opportunities
Unified Extensible Firmware Interface
  ○ Becomes NON-extensible
(Some) UEFI components

- CsmVideo
- ArpDxe
- Udp6Dxe
- UsbKbDxe
- Terminal
- SnpDxe
- IpSecDxe
- UsbMouseDxe
- SBAHCI
- MnpDxe
- UNDI
- UsbBusDxe
- AHCI
- UefiPxeBcDxe
- IsaBusDxe
- XhciDxe
- AhciSmm
- NetworkStackSetupScreen
- IsaIoDxe
- USB/XHCI/etc
- BIOSBLKIO
- TcpDxe
- IsaSerialDxe
- Legacy8259
- IdeSecurity
- Dhcp4Dxe
- DiskloDxe
- DigitalTermometerSensor
- IDESMM
- Ip4ConfigDxe
- ScsiBus
- AMITSE
- Ip6Dxe
- ScsiDisk
- UefiPxeBcDxe
- SCMCORE
- Ip4Dxe
- GraphicsConsoleDxe
- ScsiBus
- HeciSMM
- Mtftp4Dxe
- CgaClassDxe
- SetupBrowser
- AINT13
- Udp4Dxe
- EhciDxe
- HECIDXE
- Dhcp6Dxe
- UhciDxe
- AMITSE
- Ip6Dxe
- UhciDxe
- DpcDxe
- Mtftp6Dxe
- UsbMassStorageDxe
- UsbBusDxe
- XhciDxe
- Legacy8259
- DigitalTermometerSensor
UEFI Components
Standard UEFI boot steps

Platform Initialization (PI) Boot Phases

OS-present App, a.k.a. exploit home
Step 1: Replace boot with Linux

“Boot Manager” -> Linux kernel

Go-based userland (u-root.tk)

We allow DXE dispatcher if ACPI and some device initialization require it. We remove most DXEs. We kexec next kernel.
Step 2: rebuild one part of UEFI

Note: only limited source available!

Note that all of SEC/PEI is binary, but DxeCore and some small number of DXEs can be rebuilt from source.
Rebuilding bits of EFI

- https://github.com/osresearch/heads/tree/nerf
- Part of Trammell Hudson’s “HEADS” work
- Allows you to build NERF images with your own kernel and initramfs
- Has shown good results on several servers
- We are making changes to build with u-root
- This all changed just a few days ago ...
Using Linux makes firmware easier!

- Single kernel works on several boards
- We used to finely tune kernel for boards
  - No longer needed
- Caveat: it is tied to the BIOS vendor
  - Because of ACPI setup
  - Steps for AMI, TianoCore differ
- What about user space?
Userspace in Go: u-root (u-root.tk)
source-based root file system

- 5.9M firmware-based initramfs that includes
  - All command source
  - All required Go compiler and package source
  - Go toolchain
- Commands compiled on first use or at boot
- About 200ms to build; 1 ms to run
- Nice from security angle since source visible
- In some cases we want only binary so ...
Can build all u-root tools into single program for compact initramfs

- File system: 1 program and many symlinks
- Use Go abstract syntax tree package to rewrite commands as packages
- Compile into one binary (takes 15s)
- Doesn’t include source code or toolchain
- Reduces footprint to 2M
- Useful when flash space is small (<5M)
Implications for startup

- Replace *all* init scripts with Go program(s)
- Do not need systemd, upstart, scripts
- Custom-built Go binary for init is very fast
- Easier to understand than sea of files
- Note: NiChrome, based on u-root, boots Chromebook to x11+browser in 5 seconds
  - See me later if you are interested in NiChrome
We’d love to have your help!

- Testing
- Improving Travis tests
- Porting
- Contributing
- Documenting
Extra slides for u-root
Outline

● Go in 60 seconds
● What u-root is
● How it all works
● Using Go ast package to transform Go
● Where we’re going
Go in 60 seconds

- New language from Google, released 2009
- Creators include Ken, Rob, Russ, Griesemer
- Not Object Oriented
  - By design, not ignorance
- Designed for systems programming tasks
  - And really good at that
- My main user-mode language since 2010
- Addictive
package main

import "fmt"

var a struct {
    i, j int
}
Go in 60 seconds

Could also say:

type b struct {
    I, j int
}
var a b

- Note declarations are Pascal-style, not C style!
- “The type syntax for C is essentially unparsable.” - Rob Pike
func init() {
    a.i = 2
}

func main() {
    b := 3
    fmt.Printf("a is %v, b is %v\n", a, b)
}
Could also say ...

fmt.Printf("%d", b)
// You can edit this code!
// Click here and start typing.
package hi

var (  
    internal int
    Exported int
)
func youCanNotCallFromOutside() { 
    fmt.Println("hi")
}

func YouCanCallFromOutside() { 
    fmt.Println("hi")
}
package main

import "fmt"

var c = func(s string) {fmt.Println("hi", s)}
func main() {
    p := fmt.Println
    p("Hello, 世界")
    c(" there")
}
Easy concurrency: 
https://goo.gl/8Qt8WK

```go
var done = 
    make(chan int)   func main() {
func x(i int) {
    go x(5) 
    fmt.Printf("%d\n", i)  <-done
    done <- 0
}
}
```
Go in 60 seconds

- Compiler is really fast (originally based on Plan 9 C toolchain)
- V 1.2 was fastest; currently at 1.9, rewritten in Go, is still quite fast
- Compile all of u-root, including external packages, in under 15 seconds
- Package syntax makes finding all imports easy
u-root

- Go-based rootfs
  - Commands/packages written in Go
  - In one mode, MAX, compiled on demand

- 1 or 4 pre-built binaries:
  - /init
  - Go toolchain -- if compiling on demand

- Type a command, e.g. rush (shell)
  - rush and its packages are compiled to /ubin and run
  - Compilation is minimal and fast (½ second)
Key idea: $PATH drives actions

- **PATH=/bin:/ubin:/buildbin**
  - /bin is *usually* empty
  - /ubin is *initially* empty
- /buildbin has symlinks to an *installcommand*
- **First time you type rush: found in /buildbin**
  - Symlink in /buildbin: rush -> installcommand
  - Installcommand runs, builds argv[0] into /ubin
    - Execs /ubin/rush
- **Next time you type rush, you run /ubin/rush**
Installcommand is built on boot

- Init builds installcommand in /buildbin
- For each d in 
  /src/github.com/u-root/u-root/u-root/cmds/*, init creates /buildbin/d -> /buildbin/installcommand
- init forks and execs rush
  - which may be compiled by the installer and run
- init: 206 lines
“U” is for “Universal”

- Single root device for all Go targets
- New architecture requires only 4 binaries
- For multi-architecture root, proper (re)arrangement of paths is needed
  - E.g., /init -> /linux_<arch>/init
Variations on u-root for embedded

- Not everyone wants source in FLASH
- Some FLASH parts are small
- Hence the root image can take many forms
- But source code never changes
  - I.e. no specialized source code for embedded
## Variations of u-root

| 4 binaries per architecture, all commands in source form, dynamic compilation, multiple architectures in one root device | Post-boot model -- i.e. local disk, nfsroot, etc. | MAX |
| More than 4 binaries per architecture: some/all commands precompiled, dynamic compilation, multiple architectures in one root image | Post-boot model where faster boot is required | |
| 4 binaries, all commands in source form, dynamic compilation, one architecture | Pre- or Post- boot model: u-root installed in firmware or local device | |
| All commands built into one binary which forks and execs each time | Usually firmware but also netboot of “kexec” image | MIN |
A deeper look at u-root “MAX”

- Standard kernel
- four Go binaries *per architecture*
  - init/build binary (part of u-root, written in Go)
    - Merged-in minimized go build tool
  - Compile, asm, link
- All *required* Go package source
- u-root source for basic commands
- in 5.9M (compressed of course! :-)

*per architecture* indicates that there are four Go binaries for each specific architecture that u-root supports.
Root structure at boot

Go package compiled on demand

U-root init binaries (only one required)

U-root source

Other source

Go source

Go toolchain
Only one architecture type required
Init builds directories, mounts, ...

- `buildbin/` directory
  - `installer binary`
  - `Directory of symlinks built by init`
  - `ubin/` directory
    - `create etc/, dev/, proc/`
    - `mknod, mount, create any needed files (e.g. resolv.conf)`

- `installcommand`
  - `rush -> installcommand`
  - `cat -> installcommand`
  - `...`

- Directory of symlinks built by init
- `Init creates required device nodes, mount points, and mounts`
Init tasks

- /ubin is empty, mount tmpfs on it
- /buildbin is initialized by init with symlinks to a binary which builds commands in /bin
- PATH=/go/bin:/bin:/ubin:/buildbin
- create /dev, /proc, /etc
- Create inodes in /dev
- mount procfs
- Create minimal /etc/resolv.conv
Running first sh (rush)

- Init forks and execs rush
- If rush is not in /ubin, falls to /buildbin/rush (symlink->installcommand) runs
- /buildbin/installcommand directs go to build rush, and then execs /ubin/rush
- And you have a shell prompt
- From rush, same flow for other programs
Using Go to write more Go

- For scripting
- For dynamically creating shells with builtins
- For creating small memory pre-compiled versions of u-root (“busybox mode”)
run { ifaces, _ := net.Interfaces()
    for _, v := range ifaces {
        addrs, _ := v.Addr
        fmt.Printf("%v has %v", v, addrs)
    }
}

● Result:
ip: {1 1500 lo up|loopback} has [127.0.0.1/8 ::1/128]
ip: {5 1500 eth0 fa:42:2c:d4:0e:01 up|broadcast} has [172.17.0.2/16 fe80::f842:2cff:fed4:e01/64]

● But it’s not really a program … how’s that work?
‘Run’ command rewrites fragment and uses the go import package

- run reads the program
  - If the first char is ‘{‘, assumes it is a fragment and wraps ‘package main’ and ‘func main()’ boiler plate

- Import uses the Go Abstract Syntax Tree (ast) package:
  - Parses a program
  - Finds package usage
  - Inserts go “import” statements
The result

- run program builds and runs the code
- Uses Go to write new Go

```go
package main
import "net"
import "fmt"

func main() {
    ifaces, _ := net.Interfaces()
    for _, v := range ifaces {
        addrs, _ := v.Addrs()
        fmt.Printf("%v has %v", v, addrs)
    }
}
```
Taking rewriting further

- Request for single-binary version of u-root for Cubieboard
  - Allwinner A10 --> not very fast
- Wanted to compile all u-root programs into one program
Taking rewriting further

- With the ast package, we can rewrite programs as packages, e.g. ls.go

```go
package main

var x = flag.String("l", ...)
func init() {...}
func main() {
}
```

```go
package ls

var x = flag.String("ls.l", ...)
func Init() {...}
func Main() {
}
```

- Combine all of u-root into one program
- Turning 65 programs into one: 10 seconds
What is all this good for?

- Building safer startup environments
- We can verify the root file system as in ChromeOS, which means we verify the compiler and source, so we know what we’re running
- Much easier embedded root
- Security that comes from source-based root
- Knowing how things work
But I want bash!

- It’s ok!: tinycorelinux.net has it
- The tcz command installs tinycore packages
  - `tcz [-h host] [-p port] [-a arch] [-v version]`
    - Defaults to tinycore repo, port 8080, x86_64, 5.1
- Type, e.g., `tcz bash`
- Will fetch bash and all its dependencies
- Once done, you type
  - `/usr/local/bin/bash` (can be in persistent disk)
Where to get it

github.com/u-root/u-root
Instructions on
U-root.tk
Status

- Demonstrated on 4 motherboards
- Hope to have a single Go tool to do the job in a few months
- Looking for collaborators
- While we prefer coreboot-based systems we can use u-root on UEFI-based systems via NERF
Basic builtin(s)

```
builtin \n  hi '{ fmt.Printf("hi\n") }' \n  there '{fmt.Println("there")}'
```

- Create a new shell with hi and there commands
package main

import "errors"
import "os"

func init() {
    addBuiltIn("cd", cd)
}

func cd(cmd string, s []string) error {
    if len(s) != 1 {
        return errors.New("usage: cd one-path")
    }
    err := os.Chdir(s[0])
    return err
}

- This is the ‘cd’ builtin
- Lives in /src/sh
- When sh is built, it is extended with this builtin
- Create custom shells with built-ins that are Go code
- e.g. temporarily create purpose-built shell for init
- Eliminates init boiler-plate scripts
Customize the shell in a few steps

- create a unique tempdir
- copy shell source to it
- convert sets of Go fragments to the form in previous slide
- Create private name space with new /ubin
- mount --bind the tempdir over /src/cmds/rush/ and runs /ubin/rush
- You now have a new shell with a new builtin
The new shell

- Child shells will get the builtin
  - since they inherit the private name space
- Shells outside the private name space won’t see the new shell
- When first shell and kids exit, builtin is gone
- Custom builtins are far more efficient
  - Need a special purpose shell many times?
  - You can pay the cost once, not once per exec