Key Extraction - Dumping keys from the Linux Kernel Key Retention Service

Jesson Soto Ventura
@almostjson

Introduction

Principal Security Consultant

ivision

1 month in about ~50 Minutes

How this all started…

Bought a car scan tool (not this one) and wanted to get root. I am a hacker after all.

Figured I would start by intercepting network traffic.

WTF...

Encrypted HTTP traffic… to Alibaba Cloud? Every time I open the scanner app?. Why?

As root, I can surely decrypt this right? Right?!

What’s the most secure place to store secrets?

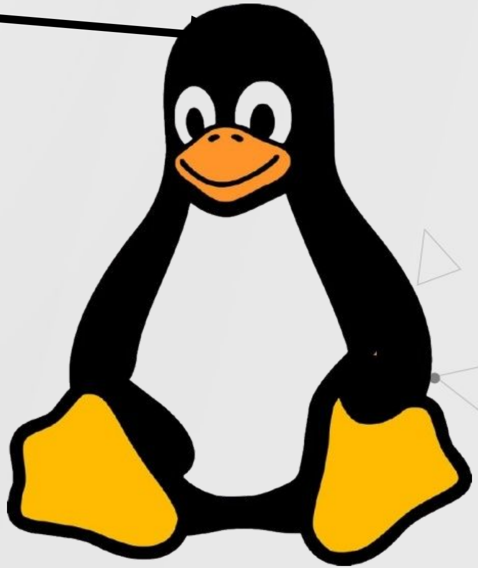
*on systems without hardware backed security modules

What’s the most secure place to store secrets?

In memory? Right?

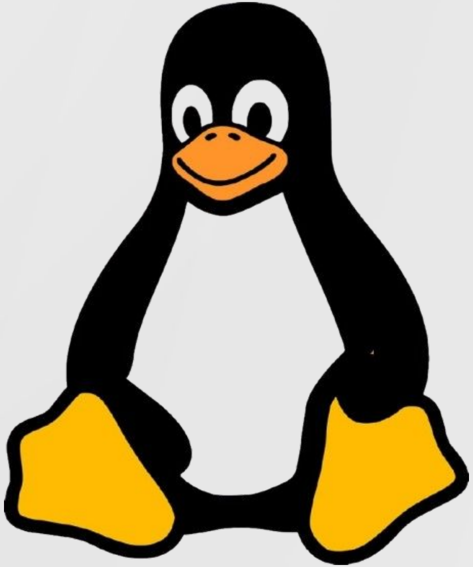
What's the most secure place to store secrets?

Yes, but not just any memory. It should be kernel memory.



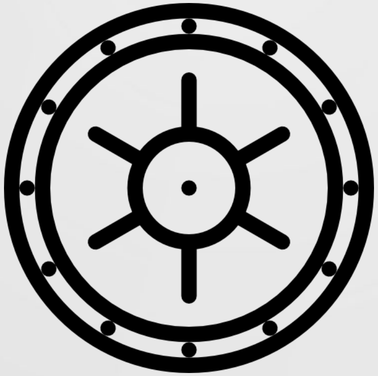
TL;DR: Linux Kernel Key Retention Service

An in-memory and in-kernel secrets vault



Let’s talk features

Secure In Kernel Key Storage



Secrets are stored in kernel memory, never written to disk.

If properly configured, secrets can not be read back even by root *without some effort*

Access Controls

Allow you to define who will have access to your secrets. There are 3 different access groups:

- User ID (UID): Limit access to only a specific user (least restrictive)
- Process ID (PID): Limit access to only a specific process
- Thread ID (TID): Limit access to a specific thread (most restrictive)

Access Controls

Allow you to define who will have access to your secrets. There are 3 different access groups:

- User ID (UID): Root can bypass this control
- Process ID (PID): Limit access to only a specific process
- Thread ID (TID): Limit access to a specific thread (most restrictive)

Access Controls

Allow you to define who will have access to your secrets. There are 3 different access groups:

- User ID (UID): Root can bypass this control
- Process ID (PID): Root can not easily bypass this control
- Thread ID (TID): Limit access to a specific thread (most restrictive)

Access Controls

Allow you to define who will have access to your secrets. There are 3 different access groups:

- User ID (UID): Root can bypass this control
- Process ID (PID): Root can not easily bypass this control
- Thread ID (TID): Root can not easily bypass this control

Also it depends on the key type

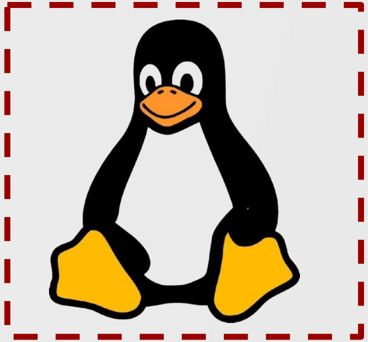
Supported Key Types

User Key

Asymmetric Key

More...

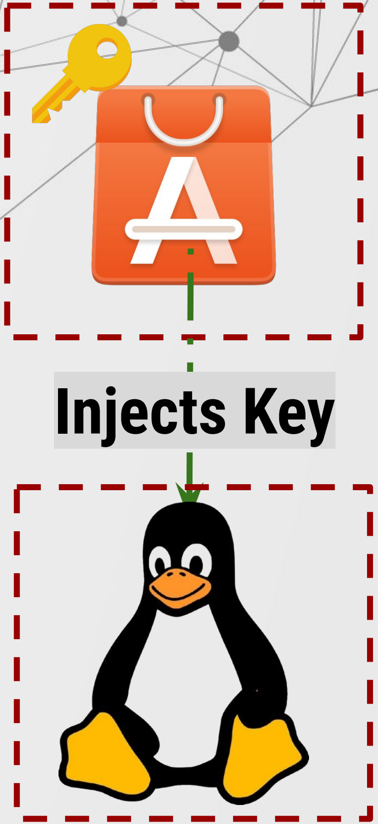
User Keys



Built for userspace applications that need to store some secret material.

Very flexible does not impose restrictions on the data stored (apart from a 64 byte limit)

User Keys - Usage Example



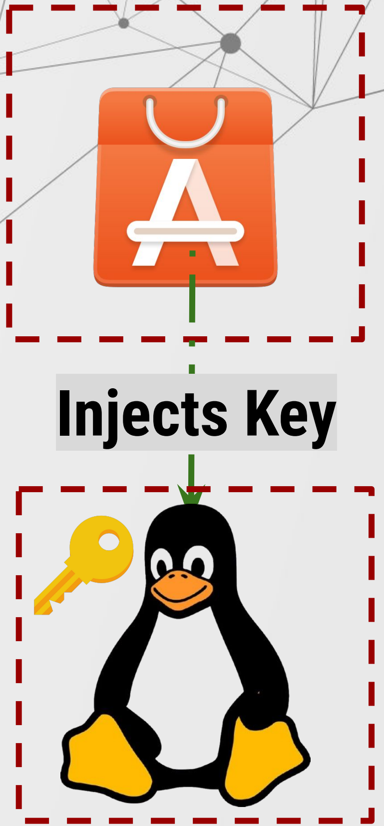
Built for userspace applications that need to store some secret material.

Very flexible does not impose restrictions on the data stored (apart from a 64 byte limit)

Typical Use Case:

1. <u>Key Injected into Kernel</u>

User Keys - Usage Example



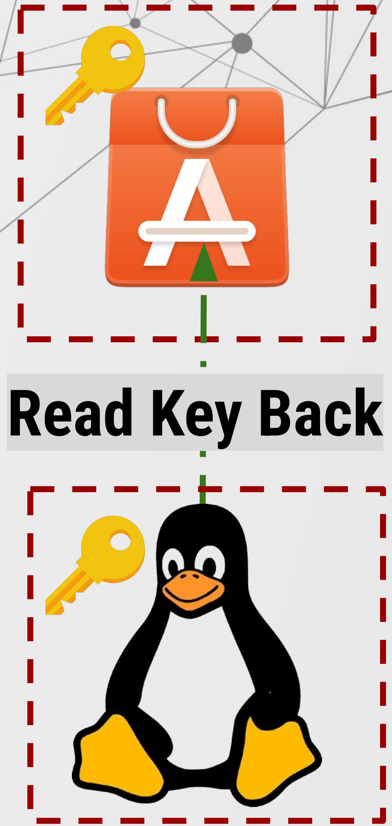
Built for userspace applications that need to store some secret material.

Very flexible does not impose restrictions on the data stored (apart from a 64 byte limit)

Typical Use Case:

1. Key Injected into Kernel
2. Key wiped from memory

User Keys - Usage Example



Built for userspace applications that need to store some secret material.

Very flexible does not impose restrictions on the data stored (apart from a 64 byte limit)

Typical Use Case:

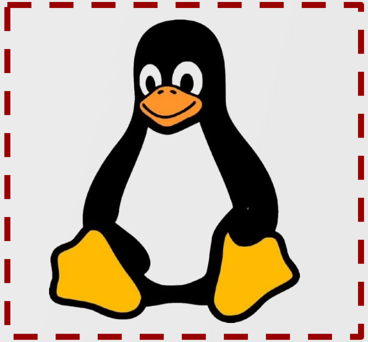
1. Key Injected into Kernel
2. Key wiped from memory
3. <u>Key read back when needed (unique feature)</u>

What if you want a more secure key type?

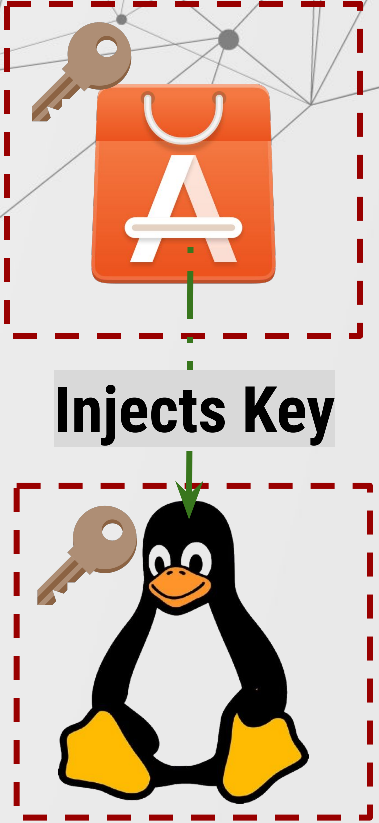
Asymmetric Key

Built for userspace applications that need to store and use asymmetric key material.

Only supports asymmetric key material



Asymmetric Key



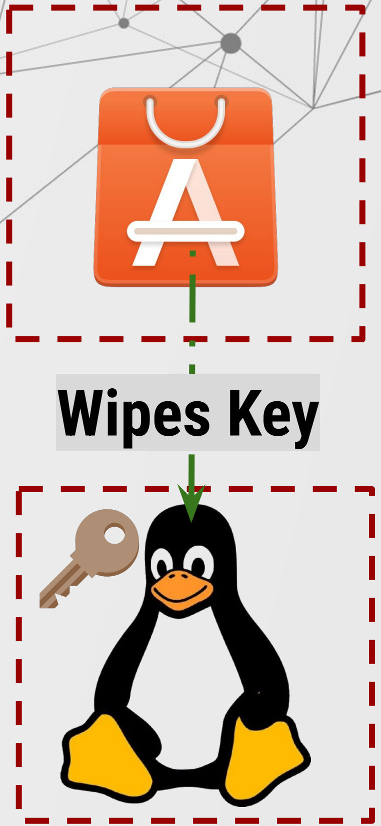
Built for userspace applications that need to store and use asymmetric key material.

Only supports asymmetric key material

Typical Usage:

1. <u>Key Injected into Kernel</u>

Asymmetric Key



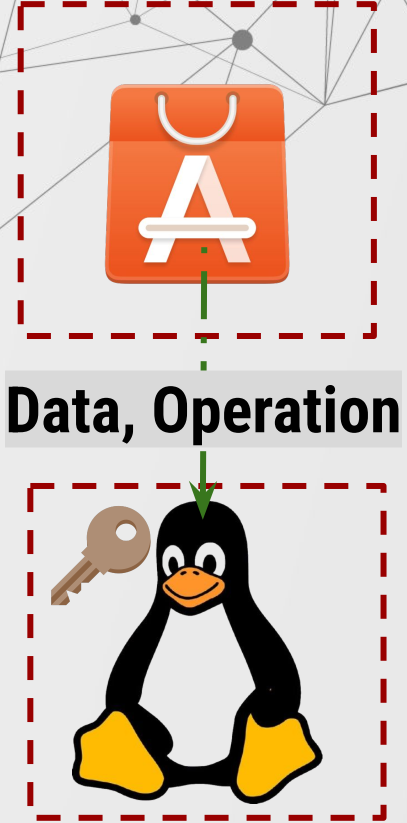
Built for userspace applications that need to store and use asymmetric key material.

Only supports asymmetric key material

Typical Usage:

1. Key Injected into Kernel
2. <u>Key wiped from memory</u>

Asymmetric Key



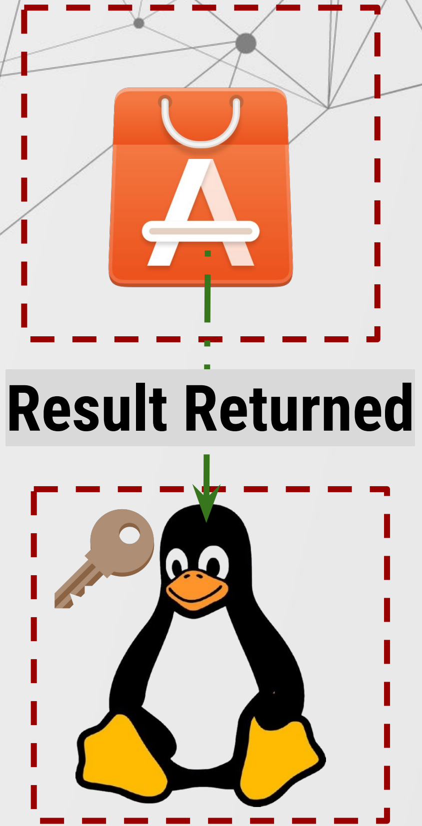
Built for userspace applications that need to store and use asymmetric key material.

Only supports asymmetric key material

Typical Usage:

1. Key Injected into Kernel
2. Key wiped from memory
3. <u>Data and operation is passed to the kernel (unique feature)</u>

Asymmetric Key



Built for userspace applications that need to store and use asymmetric key material.

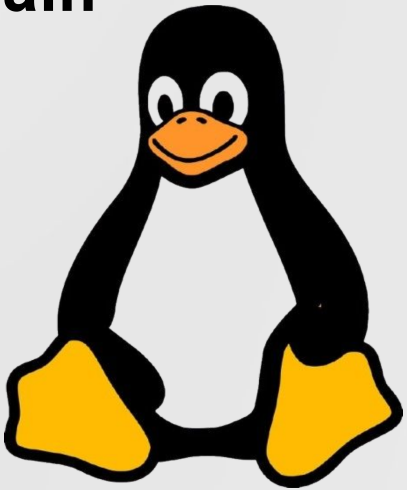
Only supports asymmetric key material

Typical Usage:

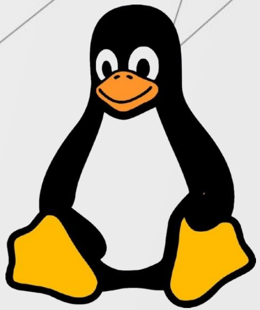
1. Key Injected into Kernel
2. Key wiped from memory
3. Data and operation is passed to the kernel (unique feature)
4. <u>Data returned by the kernel</u>
5. <u>Key can never be read back</u>

TL;DR: Linux Kernel Key Retention Service

An in-memory and in-kernel secrets vault with support for various key types and features that make key recovery a pain

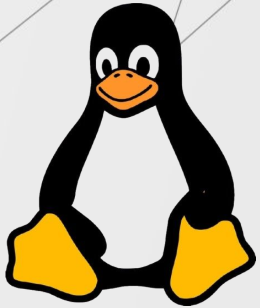


But first… we need to do some setup



You are root

But first... we need to do some setup



You are root

Injected Secrets:

Key Name	Key Data
USER_UID_EASY	_USER_UID_EASY_KEY_
USER_PID_MED	_USER_PID_MED_KEY_
USER_TID_HARD	_USER_TID_HARD_KEY_
ASYM_PUBLIC_IMP	<SOME x509 PUBLIC KEY>
ASYM_PRIVATE_IMP	<SOME PKCS8 PRIVATE KEY>

Why is it easy to dump?

<u>**User**</u>_UID_EASY

User keys support read back

Why is it easy to dump?

User_<u>UID</u>_EASY

User keys support read back

UID restrictions can be bypassed since root can be any user

Keyctl - Linux Key Management Utilities

```
Keyctl show
 228828161 --alswrv   1000  1000  keyring: _ses
  32213196 --alswrv   1000 65534   \_ keyring: _uid.1000
 605491227 --alswrv   1000  1000   \_ user: USER_UID_EASY
 256961579 --als--v   1000  1000   \_ asymmetric: ASYM_PRIVATE_IMP
 684357705 --als--v   1000  1000   \_ asymmetric: ASYM_PUBLIC_IMP
```

Show: Shows all the keys we have access to. Notice that *_MED, *_HARD, are missing because we don't have access to them

Keyctl - Linux Key Management Utilities

```
Keyctl show
 228828161 --alswrv   1000  1000  keyring: _ses
  32213196 --alswrv   1000 65534   \_ keyring: _uid.1000
 605491227 --alswrv   1000  1000   \_ user: USER_UID_EASY
 256961579 --als--v   1000  1000   \_ asymmetric: ASYM_PRIVATE_IMP
 684357705 --als--v   1000  1000   \_ asymmetric: ASYM_PUBLIC_IMP
```

Unique Key Serial: All keys are defined by a unique serial id

Keyctl - Linux Key Management Utilities

keyctl print 605491227 (USER_UID_ EASY)
_USER_UID_EASY_KEY_!

Keyctl - Linux Key Management Utilities

```
keyctl print 256961579 (ASYM_*)
keyctl_read_alloc: Operation not supported
```

That was easy! Let’s try a harder one

Dumping PID and TID protected Keys

1. Figure out which process/thread has the key.
2. Hook the process/thread using GDB
3. Wait for the key to come be read by the process
4. Profit

Dumping PID and TID protected Keys

1. **<u>Figure out which process/thread has the key.</u>**
2. **<u>Hook the process/thread using GDB</u>**
3. Wait for the key to come be read by the process
4. Profit

Gets tedious really quick when there's more than a few process or threads in use

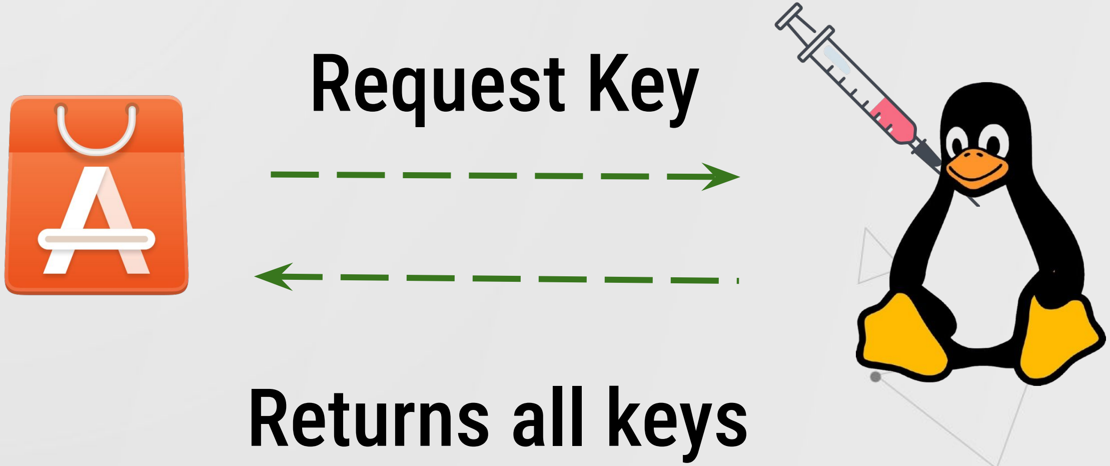
Dumping PID and TID protected Keys

1. Figure out which process/thread has the key.
2. Hook the process/thread using GDB
3. **<u>Wait for the key to come be read by the process</u>**
4. Profit

We might have to wait for a while; need to perform an unknown sequence of events to trigger the process

A better way....

Inject custom code into the kernel to bypass all controls

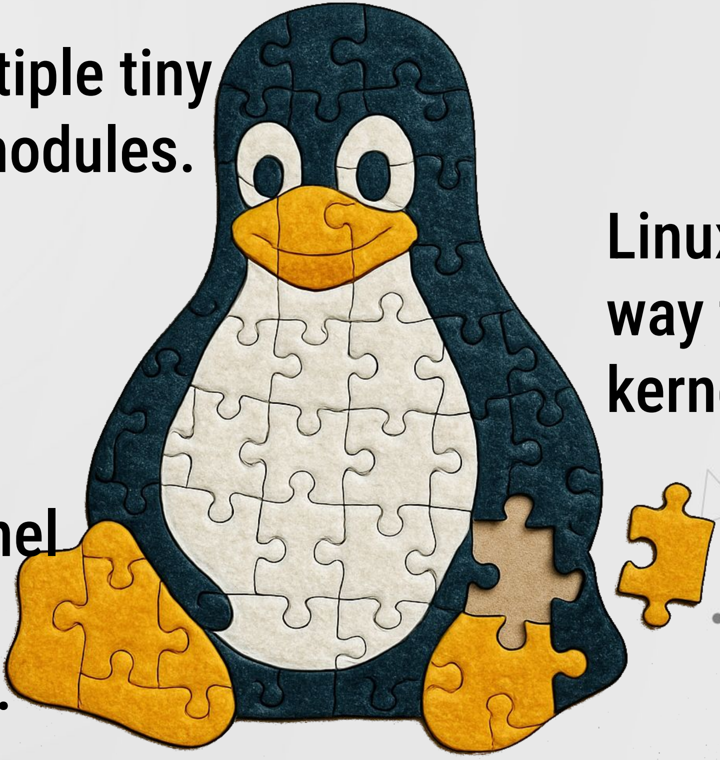


Linux Kernel Modules

Linux is made of multiple tiny components called modules.

The LKKRS is an example of a module

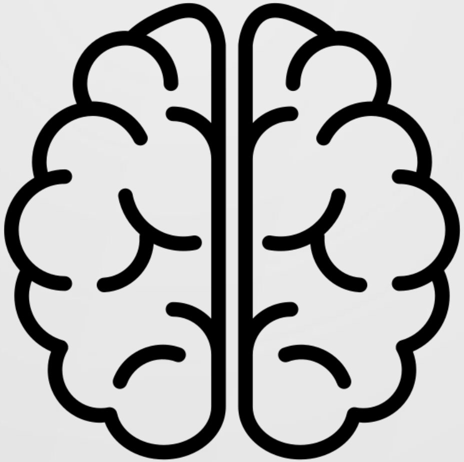
By default, Linux kernel modules
 can installed by root.



Linux kernel modules are a way to add features to kernels.

Let’s add a new feature - dump all keys

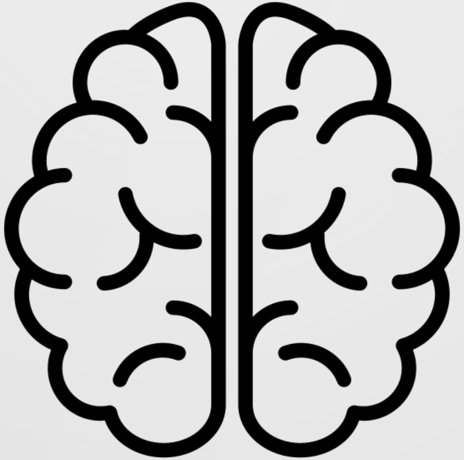
Building a Linux Kernel Module



Implementation Details

Kernel Development Headers

But first, some questions



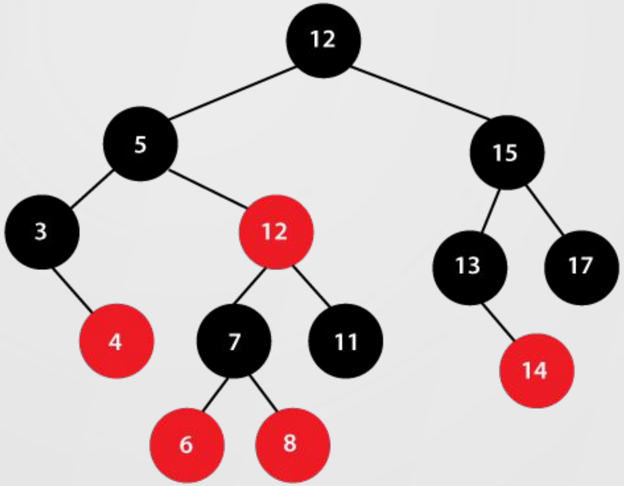
Is there a place in memory that house all keys?
Where is the key data stored?

See key.c for answers

key.{c,h}

Implementation details for the base key type class

TL;DR: Key.{c,h}



serial_key_tree: Binary tree that stores a reference to all keys.

Key blank definition (struct key), which all other keys extend and improve upon.

How it's made…

```
struct key {
…
        key_payload payload;
…
};
```

key_payload: Standard location for all key material

How it's made…

```
struct key {
…
      key_payload payload;
…
};

key_payload {
      void          *data[4];
};
```

<u>data</u>: Space for four pointers. Custom key types inject their custom key storage data structures here

Building a custom key

user_defined.{c,h}

Definition for user key type, each keytype has their own definition and implementation file

TL;DR: User_defined.{c,h}

```
struct user_key_payload {
...
    int         datalen
    char        data[64]
};
```

<u>data</u>: Actually holds our user key data. This user_key_payload struct is stored in the key_payload's data member.

TL;DR (2x): Key Recovery

User Key

key->payload.data[0]->data
Stored as raw bytes

Asymmetric Key

key->payload.data[0]->key
Stored in ASN.1 format

Time to Dump Keys!

1. Grab a key reference
2. Use it to traverse the serial_key_tree
3. Look for any user keys and asymmetric keys
4. Navigate to the key material data field
5. Print out the key’s content

Let’s build!

No headers found :doh:

Why headers matter… and getting them

Kernel Development Headers

Linux Kernel Modules have to match your exact config, or they probably won't run. There's a number of runtime check.

Plus we need access to symbols. The headers provide that access.

Easy to download on mainline linux distros:

Package_manager install linux-headers

Code Goes Here.

sotoventura.com/cackalackycon

That’s all folks…

Except not really, remember that car scan tool?

There's always something

This is an embedded system.

Embedded systems typically have custom kernels configurations and do not use mainline linux distributions

Embedded systems do not typically ship with development headers.

Getting development headers for embedded systems is difficult

Remember this?

Why headers matter… and getting them

Kernel Development Headers

Linux Kernel Modules have to match your exact config, or they probably won't run. There's a number of runtime check.

Plus we need access to symbols. The headers provide that access.

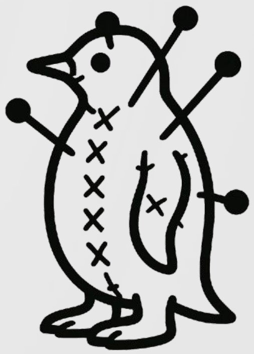
Easy to download on mainline linux distros:

Package_manager install linux-headers

Part 2: Building Linux Kernel Modules Without Headers

This is a lot of work for secrets. Guess the LKKRS is kind of effective

Bypassing Linux Runtime Checks



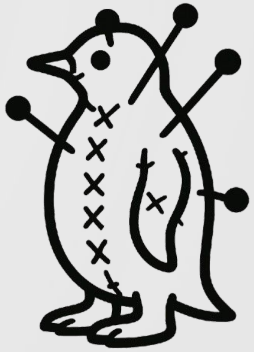
Type

What file type are Linux Kernel Modules?

file keydumper.ko
keydumper.ko: ELF 64-bit LSB relocatable, x86-64

Relocatable ELF:
gcc -c <FILE>

Bypassing Linux Runtime Checks



Type

Version

Vermagic - Linux Kernel's Magic

- A magic string in the Linux Kernel that verifies if a kernel module was compiled for a specific kernel version.
- At runtime the magic string is checked.
- All kernel modules must have this.

Copy it.

1. Find a .ko file (find / | grep .ko)
2. strings *.ko | grep vermagic
3. vermagic=6.8.0-59-generic SMP preempt mod_unload

Linux kernel headers might not be present but there's almost a guaranteed chance there's at least one linux kernel module

Copy it.

1. Find a .ko file (find / | grep .ko)
2. <u>strings *.ko | grep vermagic</u>
3. vermagic=6.8.0-59-generic SMP preempt mod_unload

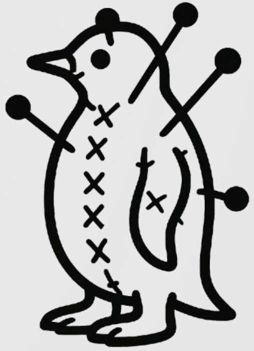
Vermagic is the same across all modules so we can just copy it

Copy it.

1. Find a .ko file (find / | grep .ko)
2. strings *.ko | grep vermagic
3. **vermagic=6.8.0-59-generic SMP preempt mod_unload**

There's a space at the end here probably… I spent way to long debugging issues because I forgot the space :facepalm goes here:

Bypassing Linux Runtime Checks

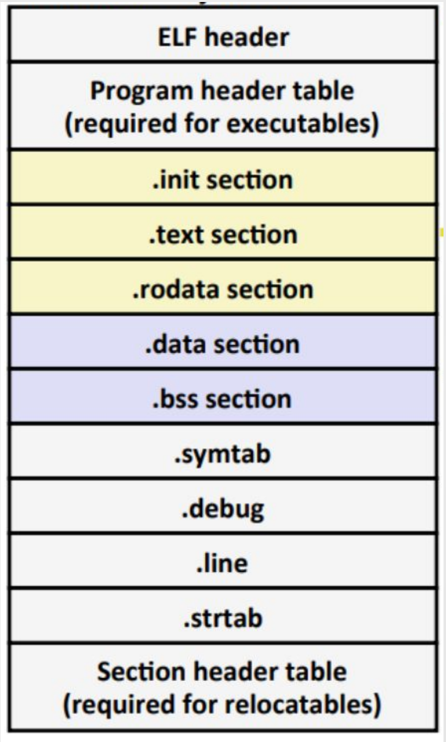


Type

Version

Format

Special Elf Section Headers



Typical ELF Headers

We’re missing two:

- .modinfo: Contains generic module information and the vermagic
- .gnu.linkonce.this_module: Contains pointers to init and cleanup functions

readelf -j .modinfo dump.ko

```
Hex dump of section '.modinfo':
  0x00000000 76657273 696f6e3d 312e3000 64657363 version=1.0.desc
  0x00000010 72697074 696f6e3d 44756d70 73204b65 ription=Dumps Ke
  0x00000020 79732046 726f6d20 4c696e75 78204b65 ys From Linux Ke
  0x00000030 726e656                          00 61757468 rnel Module.auth
  0x00000040 6f723d4                          6f 746f2056 or=Jesson Soto V
  0x00000050 656e747                          6e 73653d47 entura.license=G
  0x00000060 504c007                          6f 6e3d3439 PL.srcversion=49
  0x00000070 4239453                         236 37413633 B9E5D562F1B67A63
  0x00000080 42333539                       6e64 733d0072 B3595.depends=.r
  0x00000090 6574706f                       006e 616d653d etpoline=Y.name=
  0x000000a0 64756d70 33007665 726d6167 69633d36 dump3.vermagic=6
  0x000000b0 2e382e30 2d35382d 67656e65 72696320 .8.0-58-generic
  0x000000c0 534d5020 70726565 6d707420 6d6f645f SMP preempt mod_
  0x000000d0 756e6c6f 6164206d 6f647665 7273696f unload modversio
  0x000000e0 6e732000                            ns .
```

Key Value Pair
Data Structure
Separated by
NULL Bytes

readelf -j .modinfo dump.ko

```
Hex dump of section '.modinfo':
  0x00000000 76657273 696f6e3d 312e3000 64657363 version=1.0.desc
  0x00000010 72697074 696f6e3d 44756d70 73204b65 ription=Dumps Ke
  0x00000020 79732046 726f6d20 4c696e75 78204b65 ys From Linux Ke
  0x00000030 726e6[illegible] 61757468 rnel Module.auth
  0x00000040 6f723[illegible] 746f2056 or=Jesson Soto V
  0x00000050 656e7475 72[illegible]63656e 73653d47 entura.license=G
  0x00000060 504c0073 72[illegible] 7273696f 6e3d3439 PL.srcversion=49
  0x00000070 42394535 44353632 46314236 37413633 B9E5D562F1B67A63
  0x00000080 42333539 35006465 70656e64 733d0072 B3595.depends=.r
  0x00000090 6574706f 6c696e65 3d59006e 616d653d etpoline=Y.name=
  0x000000a0 64756d70 33007665 726d6167 69633d36 dump3.vermagic=6
  0x000000b0 2e382e30 2d35382d 67656e65 72696320 .8.0-58-generic
  0x000000c0 534d5020 70726565 6d707420 6d6f645f SMP preempt mod_
  0x000000d0 756e6c6f 6164206d 6f647665 7273696f unload modversio
  0x000000e0 6e732000                            ns .
```

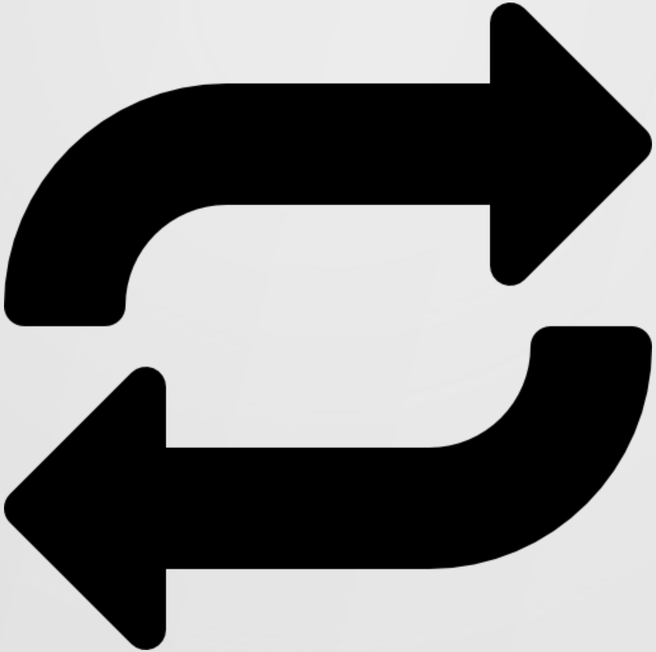
Also Vermagic is in here

How to create custom section?

const char modinfo[100] __attribute__((section(".modinfo"))) = DATA

Compiler flag telling the compiler to store variable data in a specific section rather than in its standard section

Repeat



Repeat, with two extra steps:

1. Need to make sure the section size matches
2. It has to hold two pointers (init and cleanup) at specific locations

Finding a section's size

```
readelf -W -S dump3.ko| grep .gnu.linkonce.
  [21] .gnu.linkonce.this_module PROGBITS 0000000000000000
000700 000500 00  WA  0   0 64
```

The full command has headers so you can tell which cell is the size

Finding the offsets

```
readelf -r -S ./dump3.ko| grep init -A 1
000000000138  002f00000001 R_X86_64_64
0000000000000230 init_module + 0
000000000490  002d00000001 R_X86_64_64
0000000000000260 cleanup_module + 0
```

The full command has headers so you can tell which cell is the the offset.

Creating a section with this info...

```
struct module {
    char __padding[GNU_LINK_NAME_OFFSET];
    char name [sizeof(NAME)];
    char __padding1[INIT_LOCATION-GNU_LIN
    void *init;
    char __padding2[CLEANUP_LOCATION-INIT
    void *cleanup;
    char __padding3[GNU_LINK_SIZE-CLEANUP
}__attribute__((packed));

struct module tmp __attribute__ ((section (".gnu.linkonce.this_module"))) =
{
  .init = init,
  .cleanup = cleanup,
};
```

Create a struct matching the size we need and define pointers at important offsets

Add the struct

Wait Slow down!

Headerless_kernel.py that helps you build headerless_kernel modules.

It generates a C file with all the necessary information, all you need to do is give it a working .ko file

One more problem…

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/key.h>
#include <linux/string.h>
#include <linux/key-type.h>
#include <linux/rbtree.h>
#include <keys/user-type.h>
#include <crypto/public_key.h>
#include <keys/asymmetric-subtype.h>
```

This was in our code. These includes typically rely on development headers we don’t have access too.

We need a work around so we can access functions and types

Accessing functions and symbols

```
cat /proc/kallsyms | grep printk
ffffffff9a1b3db0 T _printk
```

All kernel exported symbols, can include globals and functions. Results may vary…

Accessing functions and symbols

```
cat /proc/kallsyms | grep printk
ffffffff9a1b3db0 T _printk
```

Use extern to create your own function prototype:

```
extern void _printk (char*, ….);
```

Accessing types

No easy way, that I’ve found yet. But you do have some options:

1. Try to copy and paste the type definitions, make changes were needed. Remember everything can be a void
2. Manually recreate the data structures. This is what I do. If you do this, printk_hex_dump() is useful. You can dump live memory and stare at the bytes until they make sense

Code Goes Here.

sotoventura.com/cacklacky2025

Things to know

1. The code samples assume amd64. Changes are likely if you need to run on other architectures.
2. See my blog linked at for arm support: sotoventura.com/cackalackycon
3. Symbols exported and accessible vary by kernel. Sometimes dumping all keys is hard sometimes it’s easy.
4. There’s a few different variations on my website to help you.

So about the scan tool

It was my location, but that’s a story for another time…

Thank you

sotoventura.com

Questions ?