One-Time Pad (OTP) Implementation in the Linux Kernel

Shoukat Ali

University of Calgary

shoukat.ali@ucalgary.ca

March 20, 2020

Outline

1. One-Time Pad (OTP)
    - Background
    - Introduction
    - Characteristic of OTP
    - Example
    - Perfect Secrecy
2. User-space vs Kernel-space
    - User-space and Kernel-space
    - Why User-space?
    - Why Kernel-space?
    - Memory Isolation
    - Data encryption
3. Loadable Kernel Module (LKM)
    - Introduction
    - Making LKM
    - Simple Example
4. OTP Implementation
    - Our Implementation

Background

- G. Vernam invented a cipher in 1917 for teletype communication
- G. Vernam and J. Mauborgne (U.S. Army Captain) developed OTP
- The famous *hot line* between the White House and the Kremlin
- The pencil-and-paper versions used in diplomatic correspondence

Introduction

- Suppose $K, M,$ and C are the set of keys, messages, and ciphertexts, respectively
- In OTP cipher, encryption and decryption operations are performed as follows
    - In encryption, $M \oplus K = C$
    - In decryption, $C \oplus K = M$
    - Where $\oplus$ represents the *Exclusive OR* operation
- Encryption and decryption operations very fast

Characteristic of OTP

- The OTP key should be truly random
- The OTP key should be at least of the same length as the message
- The OTP key should be used only once
- Only two copies of the OTP key should exist
- Both copies of the OTP are destroyed immediately after use

Example

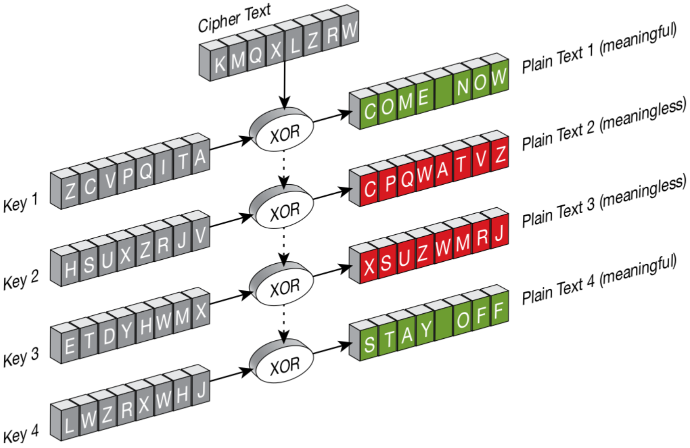

Figure: Human Language

OTP Security

- Is OTP secure?
- What is a secure cipher?
- Assume that the attacker is capable of seeing only the cipher-text

Shanon (Information-theoretic) Security

- Idea: Cipher-text should not reveal any information about the plain-text/message

Shanon (Information-theoretic) Security

- Idea: Cipher-text should not reveal any information about the plain-text/message

Definition-1

A cipher has perfect secrecy if $Pr[m|c] = Pr[m]$, for all $m \in M$ and $c \in C$, where M is the set of plain-text and C is the set of cipher-text.

Shanon (Information-theoretic) Security

- Idea: Cipher-text should not reveal any information about the plain-text/message

Definition-1

A cipher has perfect secrecy if $Pr[m|c] = Pr[m]$, for all $m \in M$ and $c \in C$, where M is the set of plain-text and C is the set of cipher-text.

Definition-2

A cipher has perfect secrecy if for all $m_0, m_1 \in M$ such that m_0 and m_1 are of same length and for all $c \in C$ we have

$$Pr\big[Enc(m_0, k) = c\big] = Pr\big[Enc(m_1, k) = c\big]$$

where $k \in K$ is chosen randomly.

Proof OTP

- Using Definition-2

Proof

$\forall m, c$

$$Pr\Big[Enc(m,k) = c\Big] = \frac{\text{No. of keys in } K \text{ such that } Enc(m,k) = c}{\text{Total no. of keys in } K}$$

we know that $k \oplus m = c \implies k = m \oplus c$

$$Pr\Big[Enc(m,k) = c\Big] = \frac{1}{\text{Total no. of keys in } K}$$

User-space and kernel-space

- User applications are executed in *user-space* and see a subset of machine's available resources
- An elevated system state where full access to all machine's resources is available is referred to as *kernel-space*

User-space and kernel-space

- User applications are executed in *user-space* and see a subset of machine's available resources
- An elevated system state where full access to all machine's resources is available is referred to as *kernel-space*

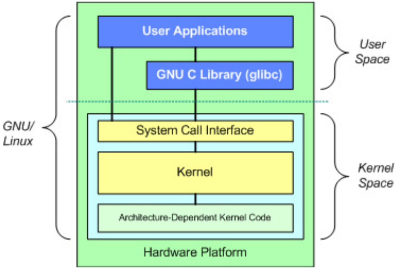

Figure: An overview of the user-space and kernel-space

User Applications

- Text editor, spreadsheet, word processing, audio and video players, web browser, etc. are some of the user applications
- Most of the applications that we use are executed in user-space

Why User-space?

- User-space programs are inherently safe because they run in protected memory

Why User-space?

- User-space programs are inherently safe because they run in protected memory
- User-space processes are not allowed to interfere with kernel memory or other user process memory

Why User-space?

- User-space programs are inherently safe because they run in protected memory
- User-space processes are not allowed to interfere with kernel memory or other user process memory
- User-space processes do not bring down the entire operating system if they crash

Why User-space?

- User-space programs are inherently safe because they run in protected memory
- User-space processes are not allowed to interfere with kernel memory or other user process memory
- User-space processes do not bring down the entire operating system if they crash
- User-space programs can easily be debugged

Why User-space?

- User-space programs are inherently safe because they run in protected memory
- User-space processes are not allowed to interfere with kernel memory or other user process memory
- User-space processes do not bring down the entire operating system if they crash
- User-space programs can easily be debugged
- But user-space processes have significant overhead when making system calls

Why Kernel-space?

- Kernel-space programs can handle interrupts

Why Kernel-space?

- Kernel-space programs can handle interrupts
- Kernel-space programs require less context switching

Why Kernel-space?

- Kernel-space programs can handle interrupts
- Kernel-space programs require less context switching
- Kernel-space programs have lower-level access to system resources

Kernel-space: Buts

- No GNU C library (glibc)
- Kernel-space programs can access the whole physical memory which implies no memory protection
- Kernel-space programs can crash the whole system
- Kernel-space programs debugging is not as easy as user application
- Kernel-space programs have no automatic clean-up

Memory Isolation

- A key security feature of modern operating systems is memory isolation
    - A user-space process cannot access other processes memory
    - User-space processes cannot access the kernel memory
- Modern processors provide supervisor bit
    - User mode: user application is in execution
    - Supervisor mode: Operating system is in execution

Data encryption

There are two way of performing data encryption

- Application level (user-space)
  - Some of the examples are; HTTPS, PGP, S/MIME, etc.
- Operating system (OS) level (kernel-space)
  - Some of the examples are; dm-crypt, encfs, IPsec, etc.

Introduction to LKM

- The Linux kernel supports Loadable Kernel Module (LKM) mechanism
- The LKM provides the following advantages
  - Loading module at run-time
  - Save kernel memory by loading module when needed and unloading when not needed
- Hence, new features/code can be added while the operating system is running

Making LKM

Some of the commands used in the making of LKM are given below

- **insmod**: to insert/load an LKM into the kernel
- **rmmod**: to remove/unload an LKM from the kernel
- **lsmod**: to list currently loaded LKMs
- **modinfo**: to display contents of *.modinfo* section in an LKM object file

A simple example

```
#include <linux/module.h>    /* Needed by all modules */
#include <linux/kernel.h>    /* Needed for KERN_INFO */

static int hello_init(void){
    printk(KERN_INFO "Hello␣World!␣I␣am␣becoming␣part␣of␣
        the␣Linux␣kernel.\n");
    return 0;
}

static void hello_exit(void){
    printk(KERN_INFO "Bye␣World!␣I␣am␣leaving␣the␣Linux␣
        kernel.\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Shoukat␣Ali");
MODULE_DESCRIPTION("The␣simplest␣kernel␣module␣");
```

Listing 1: hello_km.c

A simple example (Cont.)

- We defined two functions in the kernel module
    - `hello_init()` is invoked when the module is inserted into the kernel
    - `hello_exit()` is invoked when the module is removed from the kernel

A simple example (Cont.)

- We have used the following macros in the kernel module
  - `module_init()` specifies the function to be executed on module insertion
  - `module_exit()` specifies the function to be executed on module removal
  - `MODULE_LICENSE()` specifies to kernel the license of module and without such declaration, the kernel complains
  - `MODULE_AUTHOR()` specifies the author of module
  - `MODULE_DESCRIPTION()` specifies the functionality of module

A simple example (Cont.)

- The `printk()` function is similar to `printf()` function
- The `printk()` outputs are written in a log i.e., `/var/log/syslog`
- The `dmesg` command parses the same log
- There are eight macros in `linux/kernel.h` and in our example we have used `KERN_INFO` only which means information

A simple example (Cont.)

- To compile our module `hello_km.c`, we have
    - To create a `Makefile` in same directory of our module. Example is given in the next slide
    - Type the `make` command in terminal
    - The successful compilation will generate different files and one of them is `.ko` which represents the LKM

A simple example (Cont.)

```
obj-m:=hello.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Listing 2: Makefile

- `-C` instructs the `make` command to change the directory
- `M = $(PWD)` instructs the compiler on the source code path

Demo

- Let’s do a quick demo of what have done so far

Our Implementation

- We assume that it is possible to obtain random bytes in advance
- It is easier for adversary to attack user-space than kernel-space
- Kernel-space requires authorized access only
- Hence, kernel-space is a good place for the set K in OTP

Thanks for your attention!
Questions