Encrypted Linux x86-64 Loadable Kernel Modules (ELKM)

cenobyte
vincitamorpatriae@gmail.com
https://github.com/cenobyte-vincit/encrypted-linux-kernel-modules

***Abstract*—In this paper, we present ELKM, a Linux tool that provides a mechanism to securely transport and load encrypted Loadable Kernel Modules (LKM). The aim is to protect kernel-based rootkits and implants against observation by Endpoint Detection and Response (EDR) software and to neutralize the effects of recovery by disk forensics tooling.**

I. Introduction

For attackers, the proliferation of Linux EDR software that record `execv*()` calls make the risk of getting detected more substantial. Observable post-exploitation activity such as the execution of `insmod` to load a rootkit on a Linux host will undoubtedly catch the interest of security operations analysts hunting for threats.

In addition, there is a considerable risk of keeping LKM rootkits on the disks of compromised hosts. Rootkits need to be persistent across reboots and, on virtual machines and cloud instances it is not possible to achieve persistence through backdooring UEFI or the BIOS. Disk snapshot facilities provided by hypervisors and IaaS cloud service providers makes it trivial for adversaries to quickly and inconspicuously make a copy of a disk and recover LKMs that reside on the disk its filesystems. This risk is not only limited to virtual machines and cloud instances, as disks of physical machines can be easily imaged on the fly with `dd`, or offline using law enforcement-specific disk imaging tooling.

Threat intelligence firms receive a continuous stream of recovered artifacts from cyberattacks and while those firms are typically not that well versed in attribution, they are quite capable of analyzing these artifacts correctly. The challenge for attackers is to keep rootkits on the systems they were deployed on and to make forensics and reverse engineering as difficult as possible.

II. ELKM

The core of ELKM is the fusing of an encrypted LKM payload to an Executable and Linking Format [2] (ELF) shared object, the ‘loader’. The loader decrypts the payload during runtime, and subsequently inserts the decrypted LKM into the kernel without invoking `insmod`. As a shared object, the loader is able to hide from execution monitoring by inserting itself in dynamically linked operating system executables that are considered trusted.

ELKM supports two decryption modes:

- **Auto-decrypt**: the password is derived from the mainboard product UUID or the EC2 instance ID. In auto-decryption deployments a dropper could send the kernel version and product UUID or instance ID via a C2 channel to a staging environment. The staging environment builds the rootkit for the target kernel version, and then encrypts and fuses the payload to the loader. The dropper would then be able to fetch the staged loader and install it in an appropriate place on the host for persistence.
- **Manual decryption**: the password is provided via an environment variable or stdin.

This approach makes the deployment of ELKM-protected LKMs reasonably resilient against detection by EDR software and disk forensics tooling.

Note: Memory dumps and forensics is evidently a concern on virtual machines and cloud instances (such as Amazon EC2 [1]), but this is not necessarily the case for physical machines. The memory dump threat for attackers is outside of the scope of this paper and ELKM.

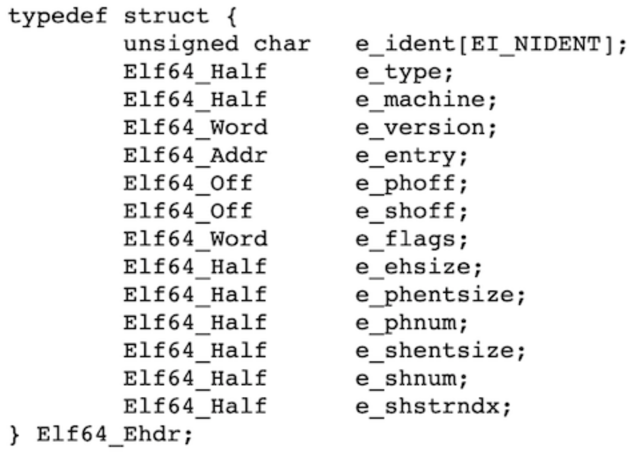
III. Loader

The ELKM loader is a 64-bit shared object. By inserting the loader using the dynamic linker and `LD_PRELOAD` on trusted executables such as systemd, crond etc. the loader can be trivially hidden from EDR software. For instance, the Carbon Black Cloud Enterprise EDR Linux sensor [3] appears to only record execution arguments, and threat hunters inspecting telemetry will not be able to see environment variables of executed programs. To overtake the execution of a target program the loader utilizes `__libc_start_main()` [4] which is a C library function that is called prior to the `main()` function.

During execution, the loader scans the value of `LD_PRELOAD` to obtain its path and then calculates the size of its ELF image to find the offset of the fused payload. Utilizing `memfd_create()` it then creates a memory file descriptor, starts decrypting the payload, and writes the decrypted bytes to the memory file descriptor. After successfully decrypting the payload in memory the next step is to load the LKM image into the kernel by using the `finit_module()` syscall. The `finit_module()` syscall reads the LKM image from the memory file descriptor and loads it accordingly.

A. Size and offset of the payload in the loader binary

The loader obtains the size and offset of the payload by reading its ELF64 [5] header. The ELF64 header structure (Fig. 1) is located at the beginning of the loader binary and is obtained using `pread64()` [6].



```
typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf64_Half      e_type;
        Elf64_Half      e_machine;
        Elf64_Word      e_version;
        Elf64_Addr      e_entry;
        Elf64_Off       e_phoff;
        Elf64_Off       e_shoff;
        Elf64_Word      e_flags;
        Elf64_Half      e_ehsize;
        Elf64_Half      e_phentsize;
        Elf64_Half      e_phnum;
        Elf64_Half      e_shentsize;
        Elf64_Half      e_shnum;
        Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

Fig. 1. Elf64 Ehdr

There are 3 elements of interest in the ELF header related to the ELF section header:

- `e_shentsize`: Holds the size of the section header's entry in bytes, all section header entries are the same size.
- `e_shnum`: Holds the number of section header entries in the section header table.
- `e_shoff`: Holds the byte offset from the beginning of the binary to the section header table.

The product of `e_shentsize` and `e_shnum` gives the section header table's size. The sum of the section header table size and `e_shoff` gives the exact size of the loader ELF image. The payload size is calculated by subtracting the ELF image size from the size of the loader binary.

B. Cryptography

The cryptographic component of ELKM is provided by OpenSSL, and it uses the Advanced Encryption Standard (AES) [7] cipher algorithm in Cipher Block Chaining (CBC) [8] mode. A 256-bit key is set, and an 8-byte cryptographically strong pseudo-random salt is generated with the `RAND_bytes()` [9] function. The `EVP_BytesToKey()` [10] function is used to derive the key and initialization vector (IV), and 10,000 iterations of the SHA-512 [11] hashing algorithm are applied to help protect against brute force attacks.

C. ELKM Payload fuser

ELKM comes with `payloadfuser` that encrypts an LKM, generates the payload, and fuses it to the ELKM loader. The resulting payload consists of two parts: the 8-byte salt and the ciphertext. The fused loader binary can be seen in Fig. 2.

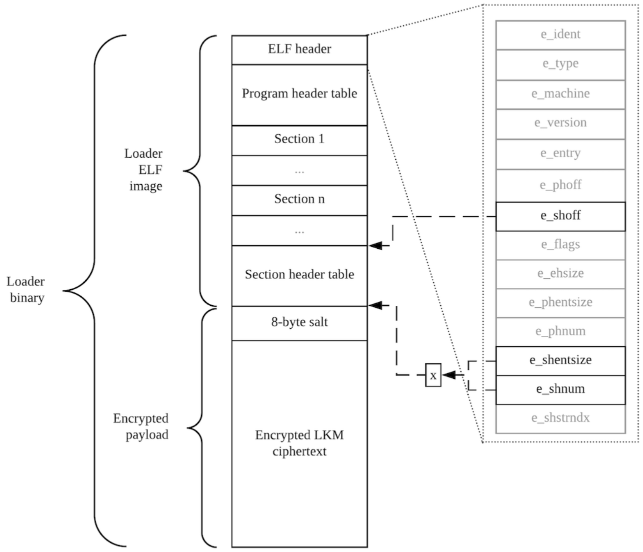


Fig. 2. Fused loader binary overview

D. Decryption password

Four methods to obtain a valid decryption password are executed in sequence:

- **Product UUID**: x86-64 Mainboards should have a manufacturer-assigned product Universally Unique Identifier (UUID) [12]. The product UUID is a unique 128-bit integer and its format is described in RFC4122 [13]. On Linux the product UUID can be obtained from `/sys/class/dmi/id/product_uuid`. The encrypted LKM can be tied to a specific host using the product UUID, and the loader will be able to auto-decrypt itself when executed on the right host. In case `/sys/class/dmi/id/product_uuid` doesn't exist, or the product UUID isn't the password, the loader will continue to check if the host is an EC2 instance.
- **EC2 instance ID**: If the host runs on Amazon EC2 then the Instance Metadata Service [15] is queried to obtain the instance ID [16]. This is considered one of the lesser secure options since the instance ID consists of a 2-character `i-` header, and a 17-character lowercase alphanumeric combination, e.g. `i-1234567890abcdef0`. If the host is not an EC2 instance, or if the instance ID is not the password, the loader will try scanning the environment next.
- **Environment variable**: The loader will use the first environment variable that has been set by leveraging glibc's non-POSIX `envp` [14] which is a list of all the environment variables that are passed as the environment of the target program. If the password is set in an environment variable other than the first environment variable, the loader will not be able to decrypt and will fall back to the last decryption option.
- **Standard input**: Interactive, which may be used post-exploitation.

IV. Protections

The ELKM loader comes with two basic protection mechanisms:

- It sets the core dump size to 0 to prevent the password from ending up in a core dump if the loader happens to terminate unexpectedly.
- It ptraces itself to prevent debuggers from attaching after a successful environment or interactive decrypt.

Future work and improvements

- Separate the loader ELF logic from the decryption logic to make it possible for a dropper to load encrypted LKMs over the network.
- The ability to tie a LKM to a single system and ask for an additional password, utilizing Cascading Multiple Block Algorithms.
- Introduce ARM and x86 support.
- Support for RHEL/CentOS 8 and Amazon Linux.

Acknowledgment

Thank you Fionn F for your review and the inspiration for writing ELKM.

References

[1] https://docs.aws.amazon.com/AWSEC2/latest/APIReference/API_SendDiagnosticInterrupt.html
[2] https://man7.org/linux/man-pages/man5/elf.5.html
[3] https://www.carbonblack.com/blog/carbon-black-cloud-adds-linux-support-for-enterprise-edr/
[4] https://refspecs.linuxfoundation.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic/baselib---libc-start-main-.html
[5] https://refspecs.linuxfoundation.org/elf/gabi4+/ch4.eheader.html
[6] https://refspecs.linuxfoundation.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/baselib-pread64.html
[7] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf
[8] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf
[9] https://www.openssl.org/docs/man1.0.2/man3/RAND_bytes.html
[10] https://www.openssl.org/docs/man1.0.2/man3/EVP_BytesToKey.html
[11] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf
[12] https://www.dmtf.org/sites/default/files/standards/documents/DSP0134_3.0.0.pdf
[13] https://tools.ietf.org/html/rfc4122
[14] https://www.gnu.org/software/libc/manual/html_node/Program-Arguments.html
[15] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html
[16] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/resource-ids.html