Malware Engineering Part 0×1 — That magical ELF

Abhinav Thakur Follow
Oct 10 · 10 min read

world is a playground

Since the day I got to hear about the term **computer virus**, I was curious to know what it exactly is or how is it created but the internet was filled with much of the random and superficial stuff about malware that it was hard to extract any useful information. Some months ago, I started with a simple file system crawler on Linux which ended up into an infector program - **Kaal Bhairav** (perhaps not a virus itself but capable of generating a few of them). The article series (discussing Kaal Bhairav) is focused on crafting a simple **segment-padded trojan** which will scan and stealthily inject Linux binaries with parasite/arbitrary code such that on execution of any infected binary by the user, the attacker's malicious code (parasite residing in host binary) is silently executed along with the binary's intended execution on the host system. This article is by no means a complete reference but does contain the core concepts which will help people getting started with binary analysis and low-level hacking domains. The article series is divided into 4 parts starting from the prerequisites and later discussing different techniques for infection -

- Part 0x1 — Focuses on parts of ELF important for our infector program.
- Part 0x2 — Focuses on implementing the first infection algorithm *(code caving the binary with parasite pill)*.
- Part 0x3 — Focuses on crafting a parasite code for our beloved host binaries.
- Part 0x4 — Focuses on implementing second infection algorithm *(extending the .text segment)*.

Prerequisites for expected audience

Before poking the devil, its better to the know the devil and his capabilities. Since we'll be dealing with file based infections on linux, an understanding of ELF file format is required for which I've prepared a hands-on course on github to understand ELF binaries (which make use of tools like xxd, readelf and objdump to dissect and understand linux binaries) or one could also read the more complete official documentation describing ELF file format (link bellow) which would greatly help understanding this article series.

- ELF binaries Binary dissection course

- ELF Specifications (Official ELF specification v1.2)
- ELF Manual (‘man elf’ on Linux)

What is an ELF binary?

Nothing related to the world of supernatural creatures but an ELF (**E**xecutable and **L**inkable **F**ormat) is the standard file format for all the binaries compiled on *NIX systems (i.e. UNIX based systems such as Linux). The ELF binaries on any *NIX system may be one of the 4 types -

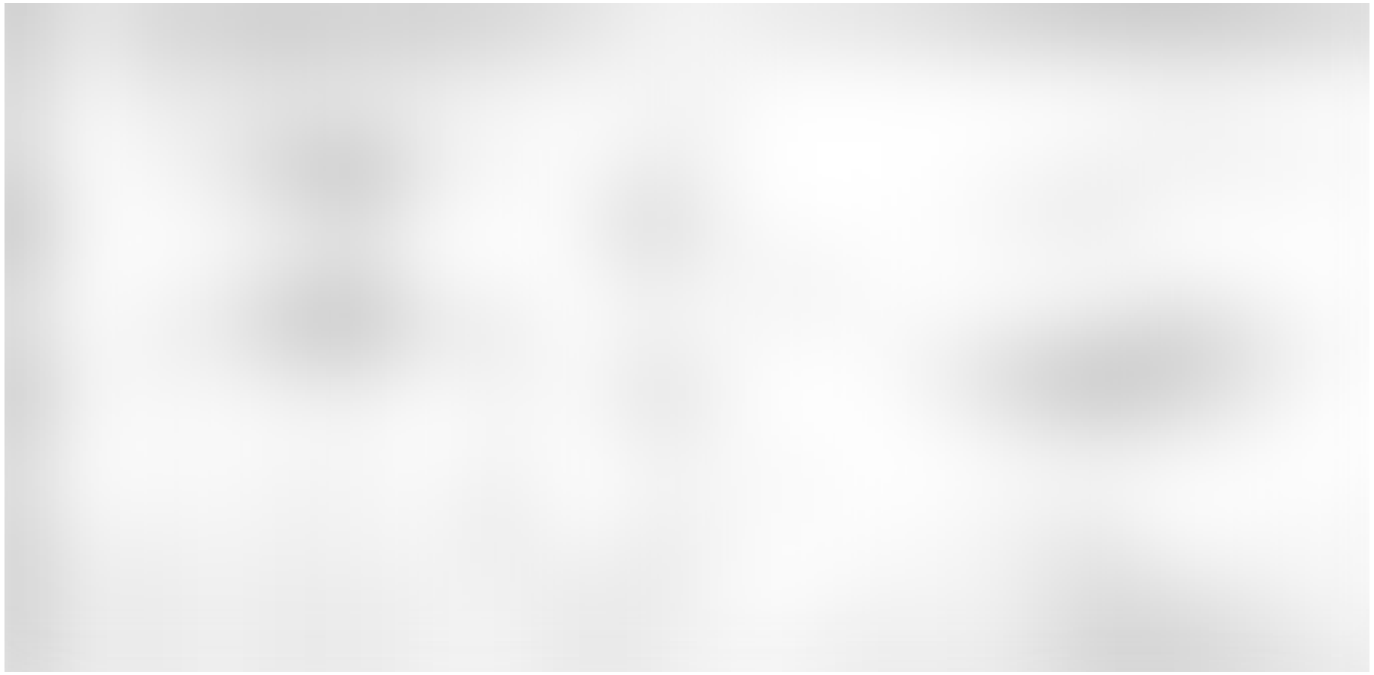
- **Executable file** (ET_EXEC) —It is linker processed, ready to execute binary. It can be compiled with `gcc -no-pie` flag on modern *NIX platforms.
- **Relocatable file** (ET_REL)— It consists of pieces of *PIC* (**P**osition **I**ndependent **C**ode) and data that is not yet linked into an executable. Conventionally named as **.o* files (object files) which are not yet processed by linker — *ld*.
- **Shared object file** (ET_DYN) — Binaries marked as ET_DYN type are shared libraries that are supposed to be loaded and dynamically linked

(at run-time) into the process address space. On modern linux distributions almost all the system binaries are compiled as shared objects (with entry point as an offset) rather executable type (ET_EXEC) in which entry point is an actual address.

- **core file** (ET_CORE) — It is a dump of program's process address space generated by the kernel when a crash happens or the program receives a SIGSEGV signal (Segmentation fault). It is useful while debugging/crash analysis of the program.

ELF skeleton, scary eh?

Looking at the bare skeleton is the best way to learn about something. Now talking about skeleton of an ELF file format, it consists of 4 main ingredients (in a nutshell)— ***ELF Header*** , ***PHT*** (short for **P**rogram **H**eader **T**able), ***Sections*** *(Actual Code & Data)* and ***SHT*** (short for **S**ection **H**eader **T**able).


ELF Skeleton

I’m explaining each of the 4 ingredients in an order suitable for explanation. You can jump to the clickable links on the headings bellow for insights or have a look on the linux manual for elf (`$ man 5 elf`) if you still don’t understand something.

Sections **—** Actual code and data used by the binary is divided into logical blocks/partitions called sections. Sections are present in the linking view (on-disk representation) of a binary only. I'll discuss briefly about some important sections here. Knowing about other sections is of course a bonus !

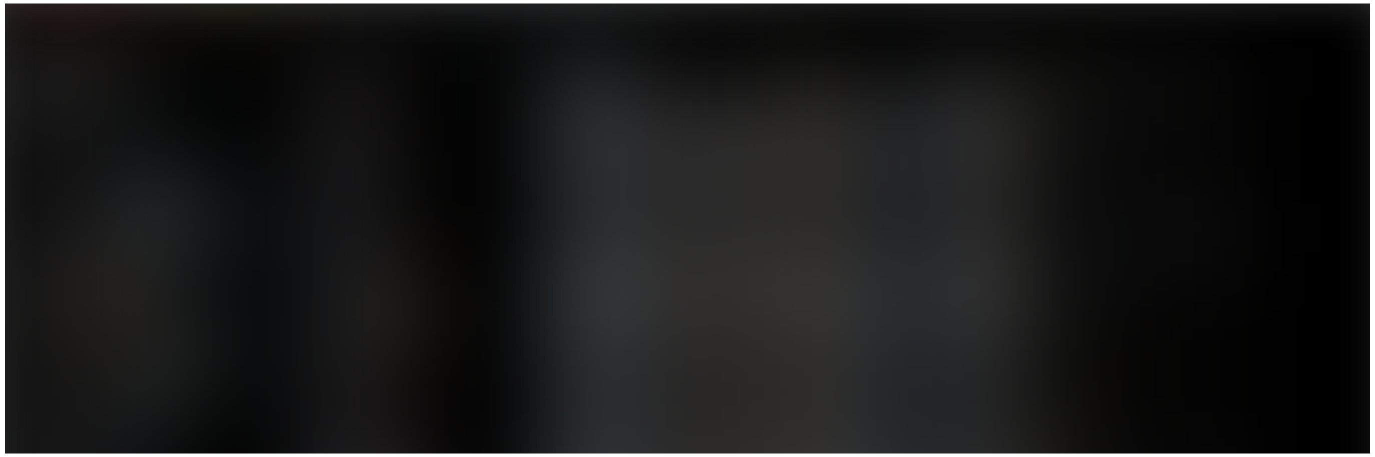
- *.init :* It contains the executable instructions which will get executed before the main function, i.e. used as the initialization code for the process.
- *.fini :* It contains the executable instructions that will get executed after the main function returns, i.e. used as the termination code for the process.
- .got : short for **G**lobal **O**ffset **T**able . It is a Table of pointers used to locate data symbols present in the binary.
- *.got.plt :* It is analogous to .got section but is used to locate library function symbols rather than data symbols in a binary. When a binary is loaded into memory, the dynamic-linker does some last minute relocations. Now, there's a process called as *Lazy binding* (*delay/lazy loading* in windows terminology) which ensures that the addresses for the shared library in .got.plt (**g**lobal **o**ffset **t**able for function symbols external to the binary) aren't resolved by the dynamic-linker until the first time a library function in invoked by the binary.

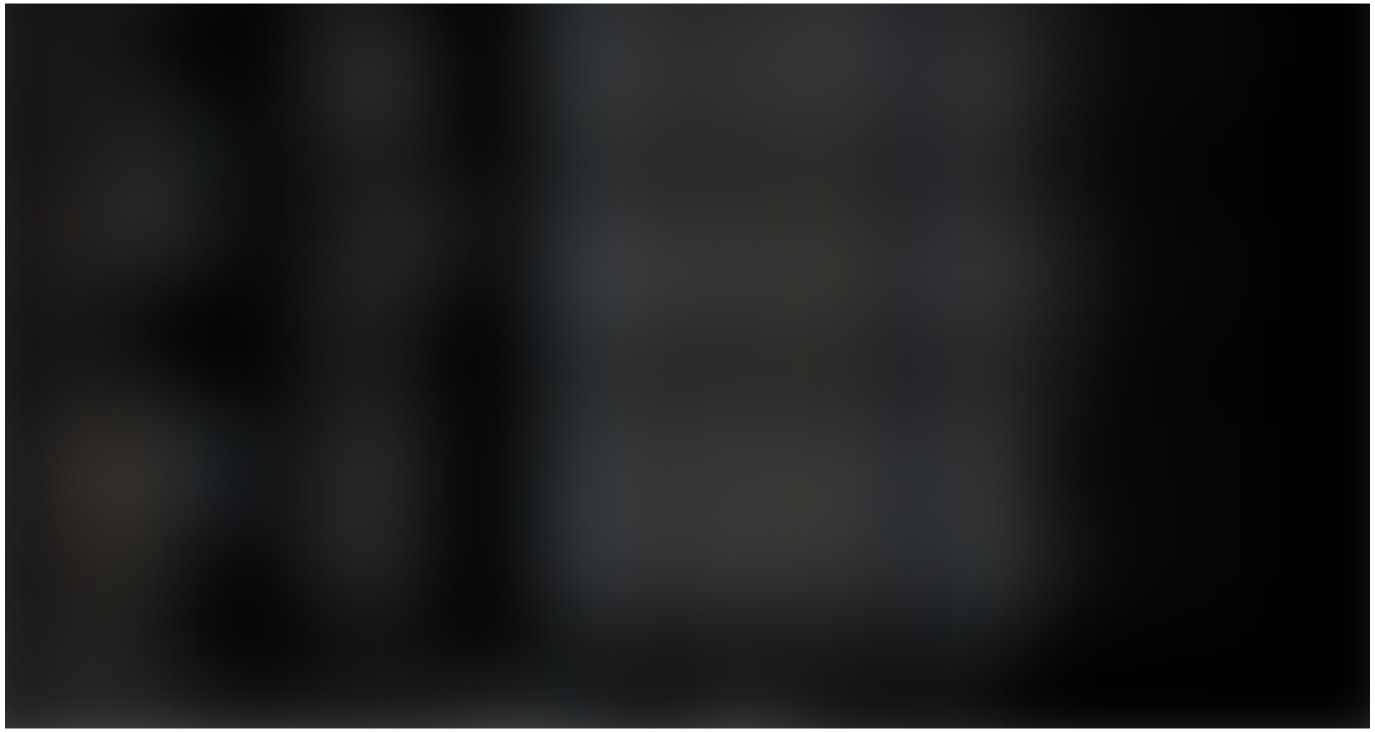
- *.plt :* Short for **P**rocedure **L**inkage **T**able. This section stores the stub-code which is used by dynamic-linker to resolve symbols (locating symbols) in the binary at run-time. It is used in conjunction with .got.plt section to perform perform *lazy-binding*.
- *.text :* All the actual executable code written by the programmer is placed in this section of the elf binary. It has access permissions of R-X (Readable & Executable but not Writable).
- *.rodata :* All initialized data is stored in this section having permissions of **R**ead-**O**nly. For eg: all string constants including any format string (provided to functions like printf/scanf etc.) while compiling will be present in this section. In the statement `printf(“Hello Hell\n”);` *”Hello Hell\n”* will be placed in .rodata section.
- *.init_array :* It stores an array of pointers(addresses) to all the code blocks (or functions) which will get executed as constructor/initialization functions for the process.
- *.fini_array :* It stores an array of pointers(addresses) to all the code blocks (or functions) which will get executed as destructor functions for the process (i.e. will get executed just after the main() exits).

- *.data :* It consists of all the initialized global and static symbols defined by your code. On program execution it is usually present in DATA segment (with **R**ead **W**rite permissions) of the process address space.
- *.bss :* It consists of uninitialized global or static symbols as defined by your code. This section is special as it does not take any space on disk (just enough space to mark its presence) and is zero initialized by its size when a program is loaded into memory. It is one of the reasons why size of file (on disk) differs from image size.
- *.symtab :* It consists of information about all the symbols present in the binary. This section is not present in stripped binaries as these symbols are not required by the dynamic linker to run the program.
- *.strtab :* This section consists of string table for the symbols present in .symtab section.
- *.dynsym :* It is present in dynamically linked binaries and consists of information about the dynamic symbols, i.e. symbols that are imported from the shared libraries or sources external to the binary. These are required by the dynamic linker at run-time and cannot be stripped from the binaries.

- *.dynstr :* This section consists of string table for the symbols present in .dynsym section.
- *.shstrtab :* This section stores all the section names (including its own name) in the form of ASCII encoded strings. This section is will be used later to find *.text* section. Use `readelf -p <section_name> <binary_path>` to dump strings from a section.

Section Header Table (SHT) — It stores the entries of all the sections where each entry describes the respective section (i.e. the section type, section size etc.) stored in the binary. The last entry in SHT is fixed *.shstrtab* section (according to what I have observed, please ping me if I'm wrong). Use `$ readelf -S --wide <binary_path>` to view SHT entries of an elf binary.



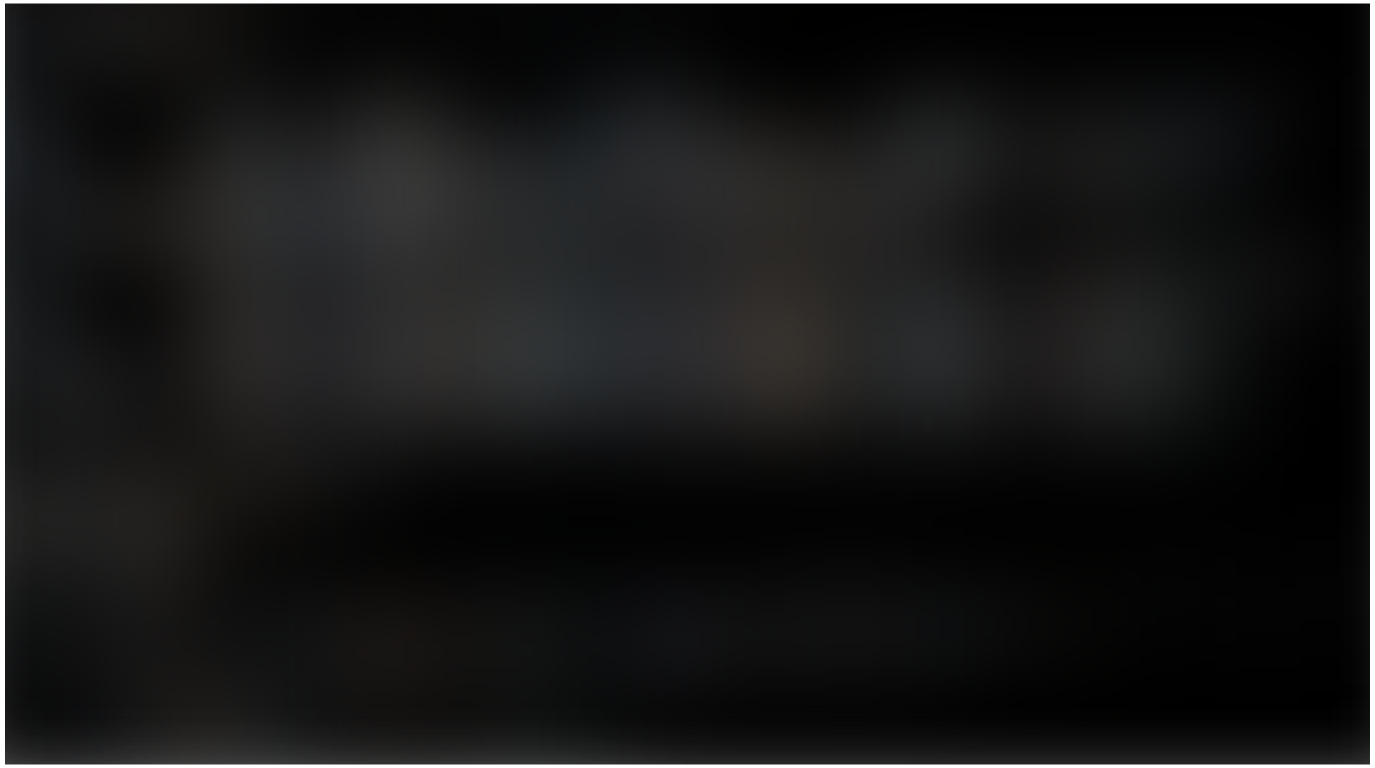

Section Header Table

- *sh_name :* It stores an index. The index acts as an offset into the .shstrtab (Section Header String Table) section which stores the section names (ASCII encoded).
- *sh_type :* This identifies the section contents and semantics. For example, PROGBITS (SHT_PROGBITS) type section holds information defined by the program, .bss is of type NOBITS (SHT_NOBITS) which means that the section occupies no space in file (on disk). See the elf manual for

details.

- *sh_addr :* The section’s first byte will appear on this address (specified by this field) if the section gets loaded into memory.
- *sh_offset :* This field stores the offset at which first byte of the section starts in file (on disk).
- *sh_size :* Size of section (on disk size).

Program Header Table (PHT) — It stores the description of segments along with sections-to-segment mappings and guides the dynamic-linker/program interpreter about how to load the on-disk representation(sections) into memory (which is divided into segments), i.e. it guides the loader on which sections will constitute what segment. If PHT doesn’t make any sense to you, you may want to read this. It is placed just after the ELF Header (i.e. at an offset of 64 bytes from the beginning of the file). Use `$ readelf -l --wide <binary_path>` to view PHT entries of an elf binary. Grep for `Elf64_Phdr` in `/usr/include/elf.h` for inner details about PHT structure.


damn these sections to segment mappings

- *p_type :* This specifies the segment type. There are generally 2 loadable segments (specified by p_type value — PT_LOAD), i.e. **code** (or .text segment) and **data segment**.
- *p_offset :* Since a segment comprises of one or more sections. It specifies the offset (from the the first byte of the file) at which the first byte of segment resides.

- *p_vaddr :* Segment virtual address.
- *p_paddr :* Segment physical address.
- *p_filesz :* Size of segment in file (on disk).
- *p_memsz :* Size of segment in memory.
- *p_flags :* Specifies the flags set on segment, i.e. **PF_R** (read), **PF_W** (write) or **PF_X** (execute).
- *p_align :* Segment alignment in memory.

There are 2 segments at **02 (CODE)** and **03(DATA)** that are marked as LOAD (i.e. space will be allocated in process memory for the sections present in these 2 segments). As we can see in *section to segment mapping* that the initialized/uninitialized global or static variables (data in .bss and .data sections) may be altered by the program code and are therefore present in the DATA segment (**03**) with **R**ead+**W**rite permissions. Similarly the .rodata section should only have Read permissions and is therefore present in CODE segment (**02**) with **R**ead+**E**xecute permissions and not in DATA segment. Also, the p_filesz and p_memsz of **03 (DATA)** segment differs due to the presence of .bss section in it (which takes 8 bytes in file to mark its presence).

ELF Header — Header consists of the metadata information about the binary or simply can say that it acts as a map to the elf binary. It consists of fields which describe what *type* of elf binary it is (shared object, executable etc), *class* of binary (x86/x86–64 bit compatibility), *metadata* pertaining to SHT/PHT etc. It occupies 1st 64 bytes and is always present at the beginning of an elf binary. Use `$ readelf -h <binary_path>` to view ELF header of an elf binary. Most Important fields include -

- *MAGIC number (e_ident[EI_NIDENT]) :* The 1st 4 bytes of a file are enough to identify a it as an ELF binary, i.e. the magic number should be 0x7f (EI_MAG0), 0x45 (EI_MAG1 or '*E*'), 0x4c (EI_MAG2 or '*L*') and 0x46 (EI_MAG3 or '*F*').
- *entry point (e_entry) :* Its either an offset (in shared object type) or an actual address (in executable type) to the _*start* symbol (from where the dynamic linker will start executing the instructions after the binary is loaded into memory). Our infector program will be using this value later to hijack control-flow of the binary.

- *Binary Type (e_type) :* Tells whether the executable is of Shared object, Relocatable, Executable or of Core type.
- *Machine Type (e_machine) :* This describes the processor-architecture for which the binary is compatible with. Our cute little parasite code (the code we'll inject into host binary) will be written according to what processor architecture is going to run the host binary.
- *SHT offset (e_shoff) :* This field in ELF Header stores the offset in file (on disk) at which the first byte of **S**ection **H**eader **T**able is placed, i.e. it tells us how far is SHT placed from the beginning (1st byte) of the binary.
- *PHT offset (e_phoff) :* This field in ELF Header stores the offset in file (on disk) at which the first byte of **P**rogram **H**eader **T**able is placed (it is usually set to 64 since PHT is stored just after ELF Header).

Other fields in ELF header which are useful while computing size of SHT/PHT will be needed while implementing SHT/PHT parsing functionality and therefore will be explained later in the article series.

. . .

EPILOGUE

Later in the series we'll see how all this information can be programmatically accessed, extracted and tampered with to craft magical spells on an ELF.
The entire source code for the elf infector can be found here.

Cheers,
Abhinav Thakur
(a.k.a compilepeace)

Github : https://github.com/compilepeace
Linkedin : https://www.linkedin.com/in/abhinav-thakur-795a96157/

Malware Hacking Cybersecurity Reverse Engineering Programming

1.3K claps

WRITTEN BY

Abhinav Thakur

security researcher

Follow

Write the first response

More From Medium

Related reads

Life as a Bug Bounty Hunter: A Struggle Every Day, Just to Get Paid

MIT Technology Review in MIT Technology…
Aug 23, 2018 · 4 min read ★

620

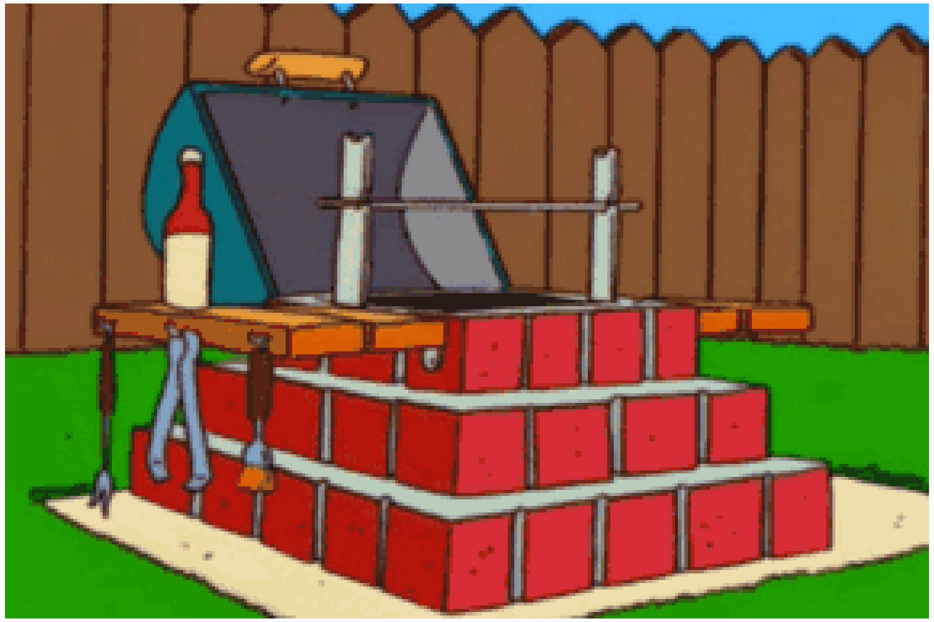
I hacked the Dutch government
and all I got was this lousy t-shirt

Related reads

Bootstrapping a Bug Bounty Program

Cheston Lee in The Startup
Sep 25 · 9 min read ★

206



Related reads

Nullcon-HackIM CTF 2019- MLAuth-Misc(500)Writeup

Aagam shah in InfoSec Write-ups
Feb 3 · 4 min read

443

mlAuth

475

Fool that mlAuth

Files

https://drive.google.com/file/d/1QvZBVns4ei1fqnhDe2uEBKlaEeR
usp=sharing

Server

http://ml.ctf.nullcon.net/predict