Relatório Técnico — Exploração *LazyFragmentationHeap* (WCTF-style)

Disciplina / Seminário: Segurança Cibernética / Exploração de Heap (CTF)
Data: (preencher data da apresentação)

Objetivo: reproduzir, entender e explorar a vulnerabilidade *LazyFragmentationHeap* conforme writeup e repositório original, documentando ambiente, metodologia, execução e recomendações.

---

1. Resumo executivo (1 parágrafo)

Este trabalho documenta a reprodução e investigação de um desafio de exploração de heap (*LazyFragmentationHeap*). Montamos um ambiente de laboratório (Kali Linux em KVM como host de ataque, Windows 10 em KVM como alvo), reimplementamos e adaptamos scripts de automação (pwntools) para warmup, grooming e triggering do componente *LazyFileHandler*, e desenvolvemos ferramentas de brute/diagnóstico que coletam dumps para análise. O objetivo prático foi obter vazamentos (leaks) úteis e, a partir deles, construir um exploit que permita execução remota. Até o momento o serviço foi alcançado, warmup/groom rodaram corretamente, o trigger foi chamado, mas os leaks não se manifestaram nas primeiras combinações automáticas — prosseguimos com uma abordagem mais agressiva (exp_bruteforce2/exp_writeup) para aumentar probabilidade de sucesso.

---

2. Ambiente de teste

2.1 Hardware / Host

- Host: Debian 13 (KVM/libvirt) — VM de desenvolvimento Kali executada via KVM.
- Máquina de ataque: VM Kali dentro do host KVM (usuário `kali`, UID 1000).
- Ferramentas no host: `docker`, `docker compose`, `virsh` (libvirt).

2.2 Máquinas virtuais

- VM atacante: Kali Linux (dentro de KVM) — contém Docker com container `lazyfrag_pwn` para rodar scripts Python.
- VM alvo: Windows 10 (KVM, NAT) — executando o binário do desafio `LazyFragmentationHeap` por meio do `AppJailLauncher`.

2.3 Rede

- Rede libvirt NAT padrão `default` (172.16/192.168.122.x).
- Endereços observados (exemplos das execuções):

    - Kali (container host) IP: `192.168.122.242`
    - Windows alvo IP (DHCP libvirt): `192.168.122.93` (observado; pode variar — fixação sugerida)
- O serviço do desafio escutou na porta TCP `6677` no Windows (visto via `netstat`).

2.4 Repositórios e referências

- Repositório oficial do desafio: `https://github.com/scwuaptx/LazyFragmentationHeap`
- Writeup: `https://null2root.github.io/blog/2020/02/07/LazyFragmentationHeap-WCTF2019-writeup.html`
- Scripts desenvolvidos / usados: `exp_auto.py`, `exp_bruteforce.py`, `exp_bruteforce2.py`, `exp_writeup.py`, `exp_full.py` (templates gerados durante a investigação).

---

3. Objetivos técnicos detalhados

1. Reproduzir o serviço alvo no laboratório e torná-lo acessível pela rede.
2. Implementar automação para:
    - warmup (preencher LFH buckets),
    - grooming (sequência de alloc/free),
    - invocação do `LazyFileHandler`,
    - coleta de dumps via `show()` e salvar localmente.
3. Detectar vazamentos (pointers, cabeçalhos PE, referências a DLLs) a partir dos dumps.
4. A partir de leaks válidos, calcular bases e offsets, montar ROP/shellcode e disparar o exploit final.
5. Documentar o processo, resultados, dificuldades e mitigação.

---

4. Metodologia e procedimentos executados

4.1 Preparação do ambiente

- Subida das VMs (Kali e Windows) via libvirt.
- Uso do `docker` no Kali para conter a ferramenta de exploração (evitar instalar toolchain diretamente na VM Kali).

- Criação de imagem Docker `lazyfrag:latest` com `python3` e `pwntools` instalado e `entrypoint.sh` que executa com UID 1000 (evita permissões root nos arquivos montados).

4.2 Acesso ao serviço e verificação

- Confirmação de conectividade:
  - `ping 192.168.122.93` — respondeu com sucesso.
  - `nc -vz 192.168.122.93 6677` — conexão TCP bem-sucedida (succeeded).
- Verificação no Windows (PowerShell) do processo e listener:
  - `netstat -ano | findstr 6677` resultou em `0.0.0.0:6677 LISTENING <PID>`.
  - Processo do launcher confirmado via `Get-WmiObject Win32_Process` (ex.: `AppJailLauncher.exe`).

4.3 Automação e primeiras tentativas

- `exp_auto.py` — script inicial para warmup + grooming + interactive:
  - Foi executado e completou warmup (80 allocs) e grooming-demo.
  - Save: `show(idxA)` gravado em `/tmp/lazyfrag_show_idxA.bin`.
  - Resultado: *sem endereços `0x...` detectados* nas saídas iniciais.
- `exp_bruteforce.py` / `exp_bruteforce2.py` — scripts de brute-forcing:
  - Implementam grid de variações: chunk sizes (0x60..0x200), free index (A/B/C), extra grooming (0..24), base (0/1), repeats, warmup sizes (80..300).
  - Geram dumps por tentativa em `/tmp/lazyfrag_brute2/...` para análise offline.
  - Resultado inicial: combinações rápidas não revelaram leaks; expandimos para brute2 agressivo (warmup 200, repeats 3), ainda sem leaks confiáveis nas primeiras execuções.
- `exp_writeup.py` — script que reproduz exatamente a sequência do writeup:
  - Permite parâmetros: warm, chunk size, extra grooming, base (0/1), repeat.
  - Está em execução no momento (conforme seu último feedback).

4.4 Observações sobre não-determinismo e tuning

- Explorações LFH são sensíveis a *timing*, *ordem* e *número* de alocações; vazamentos podem ser intermitentes.
- Recomendações adotadas: aumentar warmup (120–300), repetir inúmeras vezes, variar chunk sizes com granularidade fina, testar both 0/1-based indexing.

---

5. Resultados obtidos (resumo)

- Serviço remoto alcançável: **sim** (TCP 6677 aberto, processo ativo).
- Warmup/groom: scripts executando com sucesso e salvando dumps.
- LazyFileHandler chamado com sucesso (log e prompt do menu retornado).
- **Leak detection**: nas execuções iniciais (exp_auto test e brute combinations) nenhum leak no padrão `0x...` foi detectado.
- A investigação segue com `exp_writeup.py` e `exp_bruteforce2.py` rodando; próximos passos envolvem analisar todos os dumps gerados e insistir nas combinações que o writeup original recomenda (ou traduzir offsets do writeup).

---

6. Logs / evidências (trechos relevantes)

Abaixo seguem exemplos de comandos e trechos que você capturou — copie para anexar ao relatório:

6.1 Conectividade e listener

no Kali/container

nc -vz 192.168.122.93 6677

Connection to 192.168.122.93 6677 port [tcp/*] succeeded!

no Windows PowerShell

netstat -ano | findstr 6677

TCP 0.0.0.0:6677 0.0.0.0:0 LISTENING 4852

6.2 exec_auto.py: warmup & lazyhandler (resumo)

[*] Running quick grooming demo (warm + grooming)

[+] Opening connection to 192.168.122.93 on port 6677: Done

[+] warming LFH: 80 allocs of size 0x80

[*] Closed connection to 192.168.122.93 port 6677

[+] warmup done

[*] starting grooming demo

[*] allocated A(0), B(1), C(2)

[*] freed B (idx 1)

[*] additional grooming allocs done

[*] LazyFileHandler output:

Size:ID:Done !

****

LazyFragmentationHeap

****

...

[*] show(idxA) saved to /tmp/lazyfrag_show_idxA.bin

[*] Found addresses: []

6.3 exp_full.py / brute outputs

- Scripts de brute salvaram *múltiplos* arquivos em `/tmp/lazyfrag_brute2/...` (cada tentativa com `show_A.bin` e `lazyhandler.bin`).
- Logs de tentativa mostraram muitas combinações testadas, por exemplo:

[1] try size=0x70 free=0 extra=0 base=0 => False

[2] try size=0x70 free=0 extra=4 base=1 => False

...

Nenhum leak detectado nas combinações rápidas. Use exp_bruteforce.py para tentar mais variações ou ajuste os parâmetros.

---

7. Análise técnica (o que está acontecendo e hipóteses)

1. **Serviço funcional** — o executável aceita conexões e apresenta menu. O launcher funciona.
2. **LFH warmup OK** — alocações foram realizadas; o comportamento das funções (`Allocate`, `Edit`, `Show`, `Clean`, `LazyFileHandler`) está sendo invocado corretamente.
3. **Ausência de leak** nas primeiras tentativas — possíveis causas:
    - parâmetros de tamanho/inciso/ordem não correspondem exatamente ao cenário do writeup (diferença de versão do binário, arquitetura, ASLR/DEP/CFG variável, ou offsets diferentes);
    - necessidade de *timing* fino (pausas/ordens específicas);

- o writeup usa combinações específicas (por exemplo, liberar uma outra chunk, ou editar com tamanho diferente, ou múltiplos `LazyFileHandler` seguidos);
- ambiente Windows da VM pode ter diferenças (patches, ASLR, mitigations) que alteram comportamento;
- leak é não-determinístico e exige alto número de tentativas (rodar brute por horas).

---

8. Próximos passos sugeridos (técnicos)

1. **Executar brute mais longo** (`exp_bruteforce2.py`) durante 1–3 horas (ou até encontrar positivos). Registrar `positives_index.txt`.
2. **Executar exatamente o passo-a-passo do writeup** (se disponível) com parâmetros idênticos ao exemplo do repositório autor (eu já gerei `exp_writeup.py` para isso).
3. **Inspecionar dumps**: quando um `show_A.bin` apresentar strings `MZ`/`PE` ou `0x...`, copiar para host e analisar (`xxd`, `strings`) e me enviar para cálculo de base.
4. **Quando leak obtido**: calcular base(s) de módulo(s) e preencher `exp_full.py` com offsets/gadgets (eu faço essa etapa).
5. **Mitigação**: após sucesso da exploração, documentar mitigação. Ver resumo abaixo.

---

9. Recomendações de mitigação (para administradores de sistemas)

As vulnerabilidades de heap e fragmentação podem ser mitigadas em várias camadas:

Correção do código

- Validar tamanhos ao alocar/editar buffers; evitar referências diretas a ponteiros controláveis.
- Usar gold-standard memory APIs (secure alloc/free patterns) e checar limites.

Compilação / Proteções

- Habilitar ASLR e DEP/NX (já aplicável em Windows modernos).
- Habilitar Control Flow Guard (CFG) quando disponível.
- Compilar com /SAFESEH /GS e aplicar outras proteções específicas do compilador.

Runtime / System hardening

- Aplicar mitigations do Windows (PatchGuard, DEP, CFG).
- Reduzir privilégios do processo (rodar com conta limitada).
- Monitoramento: IDS/EDR para detectar execuções anormais; log de chamadas de API sensíveis.

Rede / Exposição

- Se a aplicação não precisa ouvir em 0.0.0.0, escutá-la em interface restrita (loopback) e usar firewall/port-forward controlado.
- Usar VPN / listas de acesso para restringir quem pode se conectar ao serviço.

---

10. Lições aprendidas e aspectos pedagógicos

- Experimentos de heap exploitation são sensíveis às condições de execução; replicação exata do ambiente do writeup é crucial.
- Automação (pwntools + scripts de brute) é essencial para acelerar a busca por condições que provoquem leaks.
- Ferramentas de contêinerização (Docker) ajudam a manter as dependências limpas e reproduzíveis durante o trabalho.
- Processo científico: formular hipóteses (por que não vaza?), testar (brute, variações), observar, documentar — um bom exercício prático para pesquisa aplicada.

---

11. Apêndices

11.1 Comandos úteis (copiar/colar)

No **Windows (PowerShell Admin)**:

```
# iniciar o launcher em background e salvar log
$job = Start-Job -ScriptBlock {
  Set-Location 'C:\Users\wctf2019\Desktop\challenge'
    .\AppJailLauncher.exe .\LazyFragmentationHeap /timeout:12000000 /key:flag.txt /port:6677 /file:magic.txt *>&1 | Out-File -FilePath C:\Users\Public\lazyfrag_run.log -Encoding UTF8
}
# ver listener
netstat -ano | findstr 6677
# visualizar log
Get-Content C:\Users\Public\lazyfrag_run.log -Tail 200
```

No **Kali / container**:

```
# entra no container como UID 1000
docker compose exec -u 1000 pwn bash -l
cd /work
```

```
# testar conectividade
nc -vz 192.168.122.93 6677

# rodar script auto/test
python3 /work/exp_auto.py test
python3 /work/exp_writeup.py run --warm 200 --size 0x80 --extra 8 --base 0 --repeat 10

# inspecionar dumps no host
docker cp lazyfrag_pwn:/tmp/exp_writeup/<tag>/show_A.bin /home/kali/Downloads/show_A.bin
xxd -l 256 /home/kali/Downloads/show_A.bin | sed -n '1,40p'
strings /home/kali/Downloads/show_A.bin | sed -n '1,120p'
```

11.2 Resumo dos scripts criados

- `exp_auto.py` — helper interativo + warmup/grooming demo.
- `exp_bruteforce.py` / `exp_bruteforce2.py` — brute-force de combinações e salvamento de dumps.
- `exp_writeup.py` — reproduz sequência exata do writeup, salva dump por tentativa.
- `exp_full.py` — esqueleto de exploit final (template), precisa de offsets/gadgets preenchidos após leak.

---

12. Conclusão

- A infraestrutura foi montada com sucesso; os scripts automatizados comunicam-se com o serviço e executam operações de warmup/groom/trigger.
- Até o momento, nos testes realizados, **não** foi obtido um leak que permita avançar automaticamente para cálculo de base e construção do ROP final.
- Continua a execução do `exp_writeup.py` e/ou do `exp_bruteforce2.py` para aumentar as chances de encontrar condições em que o leak ocorre.
- Assim que um leak com endereços hex (`0x...`) ou cabeçalho PE for obtido, eu calculo as bases e monto o `exp_full.py` final (exploit ROP + payload) pronto para execução.