Relatório Técnico — *Crypto Delphi* (UTCTF 2021)

Resumo: exploração prática de *padding oracle* em serviço que usa AES-CBC + PKCS#7; recuperação da flag `utflag{oracle_padded_oops}`; análise de impacto e mitigação segundo OWASP.

Arthur Hugo Barros Gaia

Felipe Ivo da Silva

Nathalia Cristina Santos

Thiago Zucarelli Crestani

Wilson de Camargo Vieira

Sumário

---

1 — Introdução

O desafio *Delphi* do UTCTF 2021 é um clássico de criptografia prática: um serviço aceita ciphertexts, tenta descriptografá-los com AES-CBC e devolve respostas diferentes dependendo se a validação de padding (PKCS#7) falha ou não. Essa diferença nas respostas configura um **padding oracle**, que permite a um atacante recuperar o plaintext sem conhecer a chave. A flag obtida no CTF foi `utflag{oracle_padded_oops}`.

---

2 — Objetivo do desafio

- Recuperar a flag presente no plaintext fornecido pelo serviço remoto, explorando o comportamento de erro do servidor (padding oracle).
- Demonstrar na prática como um vazamento pequeno (mensagem/erro) leva a quebra total da confidencialidade de blocos encriptados.

Referências e writeups originais (fornecidos e consultados): repositórios divulgados no enunciado e writeups públicos do CTF.

---

3 — Arquivos e writeups consultados

- Repositório com a solução e notas: `utisss/UTCTF-21/crypto-delphi`.
- Writeup complementar: `cscosu/ctf-writeups/2021/utctf/Delphi`.
- Artigos e tutoriais sobre padding oracle (fundamentação e exemplos práticos).
- OWASP — A02: *Cryptographic Failures* (Top 10 — recomendações).
- OWASP Web Security Testing Guide — *Testing for Padding Oracle*.
- Vaudenay (Eurocrypt 2002) — ataque original descrevendo limitações do CBC com padding.

---

4 — Conceitos técnicos (resumo prático)

- **AES-CBC:** encripta blocos de 16 bytes; cada bloco cifrado depende do bloco anterior (XOR com o bloco anterior após décifrar).
- **PKCS#7 padding:** esquema de padding onde N bytes com valor `0xN` são adicionados para completar o bloco. A validação do padding retorna verdadeiro/falso.
- **Padding oracle:** quando um servidor revela — diretamente (mensagem de erro) ou indiretamente (tempo de resposta, comportamento) — se o padding foi válido. Isso transforma o servidor num *oracle* que permite recuperar o plaintext bloco a bloco. A técnica é bem documentada (Vaudenay 2002) e aplicada em muitos writeups e exercícios (Cryptopals, etc.).

---

5 — Como o serviço vulnerável se comporta (observações práticas)

A partir dos writeups e das análises do serviço:

- O serviço diferencia as respostas para *padding inválido* vs *outros erros*, ou devolve mensagens/textos distintos que permitem distinguir sucesso/fracasso da validação. Esse é o sinal essencial para o ataque.
- Possível endpoint: receber hex/base64 do ciphertext e retornar uma linha indicando se a decriptação/padding foi bem-sucedida. (Implementações de CTFs com `server.py` costumam fazer isso explicitamente.)

---

6 — Estratégia de exploração — passo a passo (resumido)

1. Capturar o ciphertext alvo (IV + blocos C0,C1,...).
2. Para cada bloco `C_i` que queremos decifrar (i ≥ 1), manipular bytes de `C_{i-1}` e enviar `(IV, ..., C_{i-1}' , C_i, ...)` ao servidor.
3. Explorar o oracle: ajustar `C_{i-1}'` até que o servidor reporte *padding válido* — isso revela informação sobre `D_k(C_i)`.
4. Repetir byte a byte (do último para o primeiro) para reconstruir `P_i = D_k(C_i) ⊕ C_{i-1}`.
5. Repetir para todos os blocos relevantes até recuperar todo o plaintext (flag). (Procedimento clássico; descrição técnica e matemática no paper de Vaudenay e em tutoriais práticos.)

---

7 — Exploit completo (Python) — código comentado

> Observação: o código abaixo é um **exploit educativo** para estudar a técnica em ambiente controlado (CTF / laboratório). Não execute contra serviços sem autorização.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# ------------------------------------------------------------------------
# Solver de Padding-Oracle para AES-CBC/PKCS#7 usado no desafio "Delphi"
# Estratégia:
#   - O serviço remoto (oracle) retorna três tipos de mensagens ao receber
#     um "token" em hexadecimal:
#         * BAD  -> "Decryption failed."      (falha genérica)
#         * PAD  -> "Invalid challenge..."    (padding válido, mas challenge inválido)
#         * OK   -> "Authorization verified." (token válido, autenticação aceita)
#   - Num clássico ataque de padding-oracle, um retorno que distingue
```

```
#    "padding válido" de "padding inválido" permite recuperar o "intermediate
#    value" J = Dec_k(C) bloco a bloco. Com J recuperado, manipulamos XORs
#    para forjar IV/C1/C2/... que decifram em um plaintext desejado.
#  - Este solver é otimizado para:
#      (1) executar *batches* de 256 palpites por byte (economizando ciclos);
#      (2) recuperar J de um bloco "C_last" em uma única conexão (há ~29 ciclos);
#      (3) encadear três conexões no total para montar um token válido.
#  - Nomenclatura:
#      * Jx = Dec_k(Cx)  (valor intermediário do CBC)
#      * B1|B2 = "challenge" de 32 bytes que o servidor manda ao conectar
#      * PAD16 = bloco de 16 bytes 0x10 (padding completo)
#
# Observação didática:
#  - O CBC decripta bloco a bloco: P_i = Dec_k(C_i) ⊕ C_{i-1} (ou IV para i=0).
#  - Se controlamos C_{i-1}, conseguimos forçar o padding de P_i a ser válido
#    e, com isso, inferir bytes de Dec_k(C_i) (o "J").
#  - A cada byte resolvido, ajustamos os anteriores para "pad" consistente.
# ----------------------------------------------------------------------------

import argparse, os, socket, binascii

# Assinaturas de strings que o oracle devolve (em bytes), usadas para classificar respostas:
PROMPT = b"Please submit authorization token."
BAD    = b"Decryption failed."
PAD    = b"Invalid challenge provided."
OK     = b"Authorization verified."

def read_line(rf):
    """
    Lê exatamente uma linha do arquivo/socket encapsulado em 'rf'.
    Se não vier nada (EOF), lança exceção para que o fluxo trate desconexões.
    """
    line = rf.readline()
    if not line:
        raise EOFError("EOF do oracle")
    return line

def read_until_challenge(rf):
    """
    Ao conectar, o serviço envia várias linhas até publicar a 'Challenge: <hex>'.
    Esta função consome as linhas até encontrar a challenge e a retorna em bytes.
    """
    chall = None
    while True:
        line = read_line(rf)
        if line.startswith(b"Challenge: "):
            h = line.split(b":",1)[1].strip()
            chall = bytes.fromhex(h.decode())
            break
    return chall

def drain_until_prompt(rf):
```

```
    """
    Após enviar um batch de tentativas, o oracle imprime várias linhas,
    incluindo contadores/ruídos. Esta função descarta tudo até detectar
    o PROMPT ('Please submit authorization token.').
    Serve para "ressincronizar" a leitura entre ciclos.
    """
    # Lê até ver o prompt. Ignora contadores/ruídos entre ciclos.
    while True:
        line = read_line(rf)
        if PROMPT in line:
            return

def classify_line(line):
    """
    Classifica uma linha de resposta do oracle:
      - 'bad' se for falha genérica de descriptografia (padding inválido).
      - 'pad' se for 'Invalid challenge...' (padding VÁLIDO; estrutura OK).
      - 'ok'  se for 'Authorization verified' (token aceito).
    Retorna None para linhas irrelevantes/ruído.
    """
    s = line.strip()
    if s == BAD: return "bad"
    if s == PAD: return "pad"
    if s.startswith(OK): return "ok"
    return None  # ruído

def send_batch_and_read_results(sock, rf, hex_lines, expected):
    """
    Envia um *lote* (batch) com N linhas hexadecimais (cada uma é um token a testar),
    e em seguida coleta exatamente N classificações ('bad'/'pad'/'ok'), ignorando ruídos.
    - 'sock' é o socket aberto para enviar os bytes (mais eficiente que file.write).
    - 'rf' é o file-like (buffered reader) para ler as respostas linha a linha.
    - 'hex_lines' é a lista de strings hex (sem '\n').
    - 'expected' é o número de respostas úteis esperadas (igual a len(hex_lines)).
    """
    # Envia N linhas de HEX e lê exatamente N classificações (ignora ruídos).
    payload = ("\n".join(hex_lines) + "\n").encode()
    sock.sendall(payload)
    results = []
    while len(results) < expected:
        cls = classify_line(read_line(rf))
        if cls in ("bad","pad","ok"):
            results.append(cls)
    return results

def recover_intermediate_for_block(host, port, C_last):
    """
    Recupera o 'intermediate value' do bloco alvo:
        J = Dec_k(C_last)
    usando padding-oracle.

    Como?
```

```
  - Abrimos *uma conexão* ao oracle.
  - Criamos um bloco anterior "D" (controlado por nós), pois no CBC:
        P = Dec_k(C_last) ⊕ D
    Se acertarmos D de modo que P tenha padding válido, o oracle responde 'PAD'.
  - Fazemos brute force byte a byte (de trás para frente), ajustando D
    para simular um padding de tamanho 'pad = 1, 2, ..., 16'.
  - Otimização: em vez de 256 tentativas separadas com ida/volta, enviamos
    as 256 linhas (todas as possibilidades para o byte corrente) em um *único batch*.
  - Para cada byte, coletamos 256 respostas e escolhemos a primeira que
    seja 'pad' ou 'ok' (ambas indicam padding válido).

Notas operacionais:
  - O serviço dá ~29 ciclos por conexão; recuperar um bloco consome 16 ciclos
    (um por byte), então cabe com folga em uma única conexão.
  - Ao fim de cada ciclo, drenamos até o PROMPT para manter sincronismo.

Parâmetros:
  - host, port: endereço do oracle.
  - C_last: bloco de 16 bytes que queremos "quebrar" (em geral, um bloco
    que nós escolhemos ou que vem do protocolo).

Retorno:
  - bytes(J), onde J = Dec_k(C_last).
"""
with socket.create_connection((host, port), timeout=30) as s:
    rf = s.makefile("rb", buffering=0)
    chall = read_until_challenge(rf)  # só para sincronizar; valor não importa aqui
    drain_until_prompt(rf)

    D = bytearray(16)   # bloco 'anterior' controlado (será enviado antes de C_last)
    J = bytearray(16)   # armazenará J = Dec_k(C_last)

    for idx in range(15, -1, -1):
        pad = 16 - idx
        # Para bytes já resolvidos, ajusta D[j] para que P[j] = pad (padding consistente)
        for j in range(15, idx, -1):
            D[j] = J[j] ^ pad

        # Prepara as 256 tentativas para o byte atual (em um único ciclo)
        lines = []
        base = D[:]
        for g in range(256):
            base[idx] = g
            payload = bytes(base) + C_last  # Envia D' || C_last
            lines.append(payload.hex())

        # Envia o batch (256 linhas) e lê as 256 classificações
        results = send_batch_and_read_results(s, rf, lines, expected=256)

        # Procura a primeira resposta que indique padding válido ('pad' ou 'ok')
        found = None
        for g, cls in enumerate(results):
```

```
            if cls in ("pad","ok"):
                found = g
                break
        if found is None:
            # Em condições normais, sempre haverá pelo menos um g que satisfaz o padding
            raise RuntimeError(f"Nenhum padding válido no byte {idx}")

        # A relação fundamental:
        #   Se base[idx] = g produz P[idx] = pad,
        #   então (J[idx] ⊕ g) == pad  =>  J[idx] = g ⊕ pad
        J[idx] = found ^ pad
        D[idx] = found  # opcional, meramente para manter coerência visual

        # Drena até o prompt do próximo ciclo, mantendo a conexão "em dia"
        drain_until_prompt(rf)

    return bytes(J)

def forge_and_submit(host, port):
    """
    Encadeia as três etapas para forjar um token final válido:

    Passo 1: Escolhe um C3 aleatório e recupera J3 = Dec_k(C3)
            => Em posse de J3, podemos fabricar C2 = J3 ⊕ (0x10 * 16) para que,
               ao decifrar C2, o padding final seja 0x10...0x10 (PKCS#7 completo).
               Isso garante estrutura correta para a etapa seguinte.

    Passo 2: Recupera J2 = Dec_k(C2) do bloco recém-fabricado.

    Passo 3: Abre nova conexão, recebe o challenge B1|B2 (32 bytes).
            Queremos montar um token IV|C1|C2|C3 que autentique.
            - Definimos C1 = J2 ⊕ B2  => ao decifrar C1, obteremos P1 = J1 ⊕ B1,
              e ajustando IV = J1 ⊕ B1 faremos P0 correto para o protocolo.
            - Para achar IV, precisamos primeiro recuperar J1 = Dec_k(C1) (padding-oracle de C1).
            - Com J1 em mãos: IV = J1 ⊕ B1.
            - Token final: IV|C1|C2|C3. Enviamos em uma linha HEX e lemos a flag.

    Observações:
      - Reutilizamos a MESMA conexão do Passo 3 para recuperar J1 (16 ciclos),
        pois após o challenge o serviço permite enviar múltiplos testes até novo prompt.
      - 'pad' e 'ok' são indistintos para o propósito de "padding válido"; 'ok' pode
        aparecer ao acaso durante o brute-force, mas não atrapalha a inferência.
    """
    # Passo 1: escolha C3 e recupere J3 em conexão A
    C3 = os.urandom(16)
    J3 = recover_intermediate_for_block(host, port, C3)
    PAD16 = bytes([16])*16
    C2 = bytes(x ^ y for x,y in zip(J3, PAD16))   # C2 = J3 ⊕ 0x10*16

    # Passo 2: recupere J2 = Dec(C2) em conexão B (mesma técnica do passo 1)
    J2 = recover_intermediate_for_block(host, port, C2)
```

```
# Passo 3 (final): conectar, pegar challenge (B1|B2), recuperar J1 para C1 = J2 ⊕ B2,
# montar IV = J1 ⊕ B1 e enviar IV|C1|C2|C3
with socket.create_connection((host, port), timeout=30) as s:
    rf = s.makefile("rb", buffering=0)
    chall = read_until_challenge(rf)
    assert len(chall) == 32
    B1, B2 = chall[:16], chall[16:]
    drain_until_prompt(rf)

    # Se C1 = J2 ⊕ B2, então J1 = Dec(C1) será usado para construir IV.
    C1 = bytes(x ^ y for x,y in zip(J2, B2))

    # Recuperar J1 para C1 nesta mesma conexão (16 ciclos, um por byte).
    D = bytearray(16)
    J1 = bytearray(16)
    for idx in range(15, -1, -1):
        pad = 16 - idx
        for j in range(15, idx, -1):
            D[j] = J1[j] ^ pad
        lines = []
        base = D[:]
        for g in range(256):
            base[idx] = g
            lines.append((bytes(base)+C1).hex())
        results = send_batch_and_read_results(s, rf, lines, expected=256)
        found = None
        for g, cls in enumerate(results):
            if cls in ("pad","ok"):
                found = g
                break
        if found is None:
            raise RuntimeError(f"Nenhum padding válido no byte {idx} (J1)")
        J1[idx] = found ^ pad
        D[idx] = found
        drain_until_prompt(rf)

    # Com J1 e B1, o IV que fará P0 correto é IV = J1 ⊕ B1
    IV = bytes(x ^ y for x,y in zip(J1, B1))
    token = IV + C1 + C2 + C3

    # Envia o token (uma única linha em HEX) para autenticar e, em geral, obter a flag.
    s.sendall(token.hex().encode() + b"\n")

    # Lê algumas linhas finais (o serviço pode imprimir status/flag).
    # Não classificamos aqui; apenas repassamos ao stdout.
    try:
        for _ in range(6):
            line = read_line(rf)
            print(line.decode("utf-8","replace").rstrip())
    except EOFError:
        pass
```

```
def main():
    """
    Ponto de entrada do script.
      - Argumentos opcionais: host (padrão 'oracle'), port (padrão 4356).
      - Encaminha para forge_and_submit(), que conduz as três etapas descritas.
    """
    ap = argparse.ArgumentParser(description="Padding-Oracle solver para AES-256-CBC/PKCS7 (3 conexões, batches por byte).")
    ap.add_argument("host", nargs="?", default="oracle")
    ap.add_argument("port", nargs="?", type=int, default=4356)
    args = ap.parse_args()
    forge_and_submit(args.host, args.port)

if __name__ == "__main__":
    main()
```

Notas sobre o código:

- Ajuste `HOST`/`PORT` e a função `query_server` conforme o comportamento textual real do serviço (mensagens/formatos).
- Em CTFs normalmente o writeup inclui um `solve_delphi.py` com `pwn.remote(...)` e lógica equivalente — os repositórios que você enviou contêm versões de exemplo.

---

8 — Complexidade e custo (requests)

- Para cada byte do bloco pode-se precisar, no pior caso, até 256 tentativas; para um bloco de 16 bytes isso dá até `256 × 16 = 4096` requisições por bloco no pior caso. Em prática, a média tende a ser ~128 * 16 por bloco. Essa estimativa clássica também aparece em resumos didáticos sobre o ataque.

---

9 — Análise de impacto

- **Confidencialidade:** totalmente comprometida — plaintexts (incluindo flags, tokens, cookies) podem ser recuperados.
- **Integridade:** possível, em variantes, forjar ciphertexts válidos (CBC-R), se o adversário encadear operações.
- **Disponibilidade:** ataque gera alto tráfego e pode causar logs/alertas; mas não é um DoS primário.
- **Exemplos reais:** variantes do padding oracle foram aplicadas em frameworks e protocolos no passado; mitigação e mudanças de TLS evoluíram exatamente por essas razões.

---

10 — Mitigações concretas (mapeadas ao OWASP A02 — *Cryptographic Failures*)

OWASP recomenda evitar cenários em que falhas criptográficas vazem dados e priorizar modos autenticados (AEAD). Pontos práticos:

1. **Use AEAD em vez de CBC+MAC manual:** preferir AES-GCM ou ChaCha20-Poly1305 (autenticação + encriptação). Isso elimina a validade de padding oracles.
2. **Não revele mensagens de erro detalhadas relacionadas à criptografia:** padronize respostas genéricas (ex.: “decryption failed”) sem diferenciar por causa. Detalhes: OWASP WSTG descreve testes para padding oracle e recomenda mensagens neutras.
3. **Autenticação de ciphertext antes da decriptação:** verificar MAC/HMAC (ou usar AEAD) antes de remover padding. Se a validação MAC falhar, rejeitar sem tentar remover padding.
4. **Tempo e medidas anti-timming/side-channels:** se não for possível mudar o modo, torne as respostas e tempos indistinguíveis entre erros (difícil e propenso a falhas).
5. **Revisão de protocolos proprietários e uso de bibliotecas padrão:** evite implementar primitives você mesmo; use bibliotecas e padrões bem mantidos. OWASP recomenda evitar esquemas cryptográficos obsoletos (ex.: PKCS#1 v1.5, MD5).

---

11 — Detecção e monitoramento (práticas recomendadas)

- **Logar** tentativas de ciphertexts inválidos repetidas vezes de um mesmo IP/user (padrão de scanner).
- **Alertas:** volume anômalo de requisições que resultam em “padding error” ou “decryption failure”.
- **Instrumentação:** bloquear/ratelimit por origem após N falhas por X tempo; usar WAF com regras de assinatura para ataques de padding oracle conhecidos (também monitorar padrões de requests que mudam apenas alguns bytes do ciphertext).
- **Honeypot:** prover um token/ciphertext de teste para observar exploração automatizada e capturar endereços IP/behavior.

---

12 — Lições aprendidas e recomendações operacionais

- Pequenos vazamentos informacionais (mensagens de erro) podem quebrar segurança aparentemente forte — princípio do *fail securely.*
- Migrar serviços críticos para **AEAD** e bibliotecas modernas; rever todas as rotas que manipulam dados cifrados (sessões, tokens, dados armazenados no cliente).
- Testes automatizados (fuzzing) e checklists de segurança (OWASP WSTG) devem incluir checagem de padding oracle.

---

13 — Referências (principal)

- Writeup / repositório UTCTF Delphi — `utisss/UTCTF-21/crypto-delphi`.
- Writeup alternativo: `cscosu/ctf-writeups/2021/utctf/Delphi`.
- Vaudenay, S. — *Security Flaws Induced by CBC Padding* (Eurocrypt 2002).
- *Padding oracle attack* — resumo e matemática (Wikipedia).
- NCC Group / Cryptopals tutorial — *Exploiting CBC Padding Oracles* (prático).
- OWASP — *A02: Cryptographic Failures* (Top 10 — 2021).
- OWASP WSTG — *Testing for Padding Oracle* (guia prático de testes).
- Rizzo & Duong — *Practical Padding Oracle Attacks* (WOOT/2010) — estudo sobre variantes práticas e otimizações.