Análise de Vulnerabilidade em Protocolo de Prova de Conhecimento Zero

Desafio "Prove No Knowledge" - UTCTF 2021

Augusto César do Amaral - 790753
Gabriel Costa de Lucca - 800212
Gabriel Henrique Durigan - 792089
Leonardo Rossi Dourado - 800208
Romulo Alves da Silva - 757944
Vinícius Melo da Silva - 769722
Vitor Diniz de Oliveira Cunha - 769742
Juliano Eleno Silva Pádua - 800812

7 de outubro de 2025

1 Introdução

Este documento detalha a análise e solução de um desafio de criptografia da competição UTCTF 2021, intitulado "Prove No Knowledge". O desafio consiste em autenticar-se em um serviço remoto que implementa um protocolo de Prova de Conhecimento Zero (Zero-Knowledge Proof - ZKP) sem, de fato, possuir o segredo necessário.

Explicaremos a base criptográfica do protocolo, a vulnerabilidade em sua implementação e o processo passo a passo para explorar essa falha, enganar o servidor em 257 rodadas de verificação e obter a flag.

```
+-------------------------------------------------------------+
| UTCTF 2021 - Prove No Knowledge                             |
+-------------------------------------------------------------+
| Categoria: crypto                                           |
| Autor: qopruzijf                                            |
| Data: Mar 14, 2021                                          |
| Solves: 75                                                  |
|                                                             |
| "I've been trying to authenticate to this service, but      |
|  I'm lacking enough information."                           |
|                                                             |
| nc crypto.utctf.live 4354                                   |
+-------------------------------------------------------------+
```

Listing 1: Descrição textual do desafio "Prove No Knowledge".

2 Compreendendo o Protocolo Criptográfico

O protocolo é baseado no **Problema do Logaritmo Discreto (DLP)**, um problema matemático considerado computacionalmente difícil de resolver, que serve de base para muitos criptossistemas.

O servidor fornece três valores: um gerador `g`, um módulo primo `p`, e uma chave pública `y`. Estes valores estão relacionados por uma chave secreta `x` através da equação:

$$y \equiv g^x \pmod{p}$$

O objetivo de um protocolo de ZKP neste contexto é permitir que um cliente (o provador) convença o servidor (o verificador) de que ele conhece o segredo `x` sem nunca revelar o valor de `x`. A pegadinha deste desafio é que nós, como atacantes, não conhecemos `x`.

3 Vulnerabilidade: Previsibilidade dos Desafios

A segurança de um protocolo de ZKP interativo depende fundamentalmente da aleatoriedade dos desafios enviados pelo verificador. Se o provador puder prever qual desafio será feito, ele pode preparar uma resposta forjada para passar na verificação sem conhecer o segredo.

A implementação do servidor neste desafio é falha: em vez de escolher aleatoriamente o tipo de verificação a cada rodada, ele **alterna previsivelmente** entre dois cenários.

3.1 Cenário 1: Verificação Direta (Rodadas Ímpares)

Neste cenário, o servidor nos pede para enviar um valor `C` e, em seguida, o expoente `r` que o gerou.

1. **Servidor:** "Envie-me $C = g^r \pmod{p}$ para um r aleatório."
2. **Servidor:** "Agora, revele o valor de r."

Esta verificação é trivial de passar. Nós simplesmente escolhemos um `r` aleatório, calculamos `C`, enviamos, e depois enviamos o `r` que usamos. O servidor verifica se $g^r \pmod{p}$ é igual ao `C` que enviamos, o que sempre será verdade. Não é necessário conhecer `x`.

3.2 Cenário 2: Verificação com Chave Pública (Rodadas Pares)

Neste cenário, a verificação normalmente dependeria do segredo `x`. O servidor valida a relação entre nosso valor `C`, a chave pública `y` e um segundo valor que fornecemos. A verificação do servidor é equivalente a checar a equação $g^s \equiv C \cdot y \pmod{p}$.

1. **Servidor:** "Envie-me um valor de compromisso C."
2. **Servidor:** "Agora, envie-me um valor s para a verificação."

Como não conhecemos `x`, não podemos calcular o `s` correto se tivéssemos enviado $C = g^r$. O truque é forjar o valor de `C` desde o início. Em vez de enviar g^r, enviamos:

$$C = (g^r \cdot y^{-1}) \pmod{p}$$

Onde y^{-1} é o inverso multiplicativo modular de `y`. Quando o servidor realiza a verificação, ele calcula $C \cdot y$:

$$C \cdot y \equiv (g^r \cdot y^{-1}) \cdot y \equiv g^r \pmod{p}$$

Portanto, para que a verificação $g^s \equiv g^r$ seja válida, basta que $s = r$. Assim, quando o servidor nos pede o segundo valor, nós simplesmente enviamos o mesmo `r` aleatório que usamos para forjar `C`.

4 Estratégia de Exploração e Implementação

A exploração consiste em conectar ao servidor e responder às 257 rodadas, aplicando a estratégia correta para cada uma com base em sua paridade.

- **Se a rodada for par (0, 2, 4...):** Usamos a estratégia do Cenário 2 (o truque com o inverso modular).
- **Se a rodada for ímpar (1, 3, 5...):** Usamos a estratégia do Cenário 1 (a verificação direta).

Um script em Python utilizando a biblioteca `pwntools` foi desenvolvido para automatizar este processo.

```python
from pwn import *
from sympy import mod_inverse
from random import randrange

host, port = "crypto.utctf.live", 4354

r = remote(host, port)

r.recvuntil(b"g: ")
g = int(r.recvline())
r.recvuntil(b"p: ")
p = int(r.recvline())
r.recvuntil(b"y: ")
y = int(r.recvline())

for round_num in range(257):
    r.recvuntil(b"Send")

    if round_num % 2 == 0:
        # Estrategia para rodadas PARES (Cenario 2 - O Truque)
        ran = randrange(p)
        y_inv = mod_inverse(y, p)
        C = (pow(g, ran, p) * y_inv) % p
        r.sendline(str(C))
        r.recvuntil(b"Send")
        r.sendline(str(ran))
    else:
        # Estrategia para rodadas IMPARES (Cenario 1 - O Facil)
        ran = randrange(p)
        C = pow(g, ran, p)
        r.sendline(str(C))
        r.recvuntil(b"Send")
        r.sendline(str(ran))

    print(f"Rodada {round_num} passada com sucesso!")
```

```
36
37 flag = r.recvall()
38 print("\n--- FLAG ---")
39 print(flag.decode())
```

Listing 2: Script de exploracao em Python utilizando ‘pwntools‘.

5 Resultado

A execução do script (listado em 2) completa com sucesso todas as rodadas de verificação. Ao final, o servidor nos concede a flag, que confirma a natureza da vulnerabilidade explorada.

```
utflag{questions_not_random}
```

Listing 3: Flag obtida após a execução bem-sucedida do script.

A flag, `utflag{questions_not_random}`, explicitamente declara a falha: as perguntas (desafios) do servidor não eram aleatórias, permitindo que o protocolo fosse contornado.

6 Conclusão

Esta vulnerabilidade demonstra o impacto crítico de uma implementação incorreta de um protocolo criptográfico. Embora o Problema do Logaritmo Discreto seja matematicamente seguro, a previsibilidade e a falta de aleatoriedade nos desafios do servidor criaram uma brecha que permitiu a autenticação sem o conhecimento do segredo. Isso ressalta a importância de seguir estritamente as melhores práticas de segurança na implementação de sistemas criptográficos, garantindo que a aleatoriedade, um pilar fundamental de muitos protocolos, seja preservada.