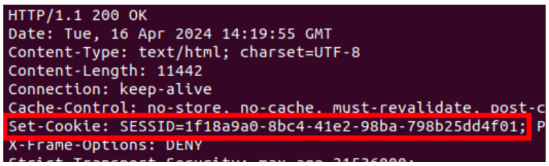
*Vulnerabilities: important case studies*

## PaloAlto Networks Global protect

- Specialized in selling devices for firewall and VPN, devices that are made to be on the border of an organization and meant to protect the internal part of the organization and to protect the remote work
- There is a web application that enables to configure the devices, that has a tiny little bug
  - Web a browser connect to an application, it maintain session, the session identifier is in a cookie
  - The structure of the cookies is shown in figure and it is in a certain correct standard, but the web app, when the application receives an HTTP request does not check the structure of the cookie




- Consequence? Unauthenticated attacker may execute arbitrary code with root privilege on the firewall, this appliances are meant to put on the border of an organization and exposed on internet

## How the appliance work (reverse engineered)

- Few details of how it works
- Session state management: session table (in which the data structure in which the server maintains the state of every session) in implemented in the file system, for every active session, in a certain directory is stored a file named /tmp/sslvpn/session_ID (state of the session in that moment)
- Telemetry data: (what happened, a summary)) is sent every hour via HTTPS to a configuration-specified URL, transmission is done by invoking the curl (you can use the command to store a file to a certain URL or download a file from an URL) command with a shell

```
...
curl_cmd = "/usr/bin/curl -v
      -H \"Content-Type: application/octet-stream\"
      -X PUT \"%s\"
      --data-binary @%s
      --capath %s
      --interface %s"\
      %(signedUrl, fname, capath, source_ip_str)
...
pansys(curl_cmd, shell=True, timeout=250)
...
```

-The web application invokes the command this way: it first constructs a string containing the command to execute, and among there is the URL in a string, a string containing the name of the file that has to be uploaded on that location. Then the app invokes a function with 3 parameters
-After several invocations there is the line that actually invokes the first command. The invocations occur with a subprocess.Popen (subprocess is a python library, Popen is a function), there are a lot a parameters, to us is important that the string of the command is passed to this library and the parameter cmd is used for a certain purpose. Basically the subprocess command means "create a shell process that executes the command"

- The file that is uploaded every hour is obtained by looking in these directories :
  - /opt/panlogs/tmp/device_telemetry/hour, /opt/panlogs/tmp/device_telemetry/day
- Whenever it finds a new file that is not been uploaded, starts the process to upload it in the specified URL

## Putting all together

- Web application does not check the cookie values to have expected syntactic structure + session state of session ID stored in file named /tmp/sslvpn/session_ID → send HTTP request with carefully structured cookie value will create an empty file with attacker-controlled name in any arbitrary attacker-chosen directory
- Bad practice:
  1. **Web applications run as roots**: not necessarily wrong, in this case essential for the exploit
     - Because since the web app runs as root, if you use as file name a path that attempts to create a file in a different directory, it will be created, because every command run by a root account is executed
     - so far we can write an empty file of out choose in telemetry directories, so curl will send an empty file to the configuration-specified URL, it is not useful yes, this is not yet a problem
- There is another problem: the subprocess.Popen library function has one parameter whose name si shell and whose name is a boolean, can be used in 2 ways:
  - if the shell parameter is set to true, the cmd parameter is a string, and the function executes a shell that executes the string
  - If the shell parameter is set to false, the cmd parameter is expected to be a list of strings, the function executes cmd[0] with remaining elements as parameters (this is the default) (in this case Popen does not open a shell)

*Vulnerabilities: important case studies*

- If you invoke Popen with shell=true, since you open a shell, cmd may be any string in the shell syntax, if its content is derived from input data then you must make sure there is no injected command (if the string is made by the program is one thing, if it is set by an input, then you have to be really careful)

2. **Executes a shell even though it needs to execute one single command**, this is also not necessarily wrong, but essential for exploit in this case

- Linux shell feature: command enclosed within backticks
  - Bash behavior: execute that command first, replace it with its output, execute enclosing command (sono tipo dei comandi tra virgolette che vengono eseguiti prima degli altri, tipo "echo "Today is 'date'", prima viene eseguito 'date', viene sostituito con la data, poi viene eseguito il comando echo)

`Cookie: SESSID=PATH_TO_TELEMETRY_DIR/aaa`command``

-So at this point the cookie management file will create a file named aaa 'command' in telemetry directory, telemetry will invoke a shell for executing curl_cmd (curl… -X PUT aaa 'command'…) so the shell will execute the command (tra virgolette quindi eseguito per primo) and then curl, all of this with root privilege

→ unauthenticated remote command injection

- Default subprocess.Popen: if it is used in default configuration, telemetry would not invoke a shell for execute curl…-X PUT aaa 'command', but curl would only upload that empty file to the configuration URL (no remote command injection)
- Trivial question: are these memory safety issues? Would a memory safe programming language have helped? No, the web app is implemented in python that is memory safe, it is not an issue of memory safety
- Never forget: web applications that do not check the cookie values to have the expected syntactic structure + two bad practices = unauthenticated remote command execution
- Understanding the vulnerability:
  - the exploit: richiesta HTTP
  - the payload: la parte che l'attaccante vuol fare eseguire, quindi la parte della richiesta che contiene il comando
  - Threat model: devo avere la possibilità di comunicare con l'applicazione (aprire connessione TCP con l'app)
  - Sviluppare l'exploit per questa vulnerabilità è banalissimo, chiunque è in grado di svilupparlo e iniettare l'exploit è altrettanto banale perché si tratta di un server esposto su internet senza nemmeno bisogno di autenticazione
  - Può essere usata per initial access, corrisponde alla tecnica dell'exploit public facing application
  - Può essere usata anche per ottenere l'execution, corrisponde alla tecnica del command and scripting interpreter
  - Può essere usata anche per ottenere persistence, avendo una shell posso fare quello che mi pare, per esempio create account, boot or logon autostart execution…

---

## Important lesson

- Unsafe practices are unsafe. Period.
- One cannot predict their reach
- avoid unsafe practices as much as you can
- Vulnerabilities never result from one single reason, they always result from many different and seemingly unrelated reasons
- Never trust input data: never, not even implicitly. The structure and values may not be what you expected (much easier said than done)
- Testing: more intensive/accurate testing might have discovered a bug, it would have not discovered this vulnerability automatically
  - Do not expect to discover a vulnerability in your testing campaigns, you will usually observe only bugs
- Bugs have to be categorized: not vulnerability, hardly exploitable vulnerability or exploitable vulnerability
  - Categorization requires skills, time and manual work
  - Never quickly dismiss a bug as "its not a vulnerability"
  - Never start crying whenever you discover a bug
- Deep question: in testing a software product, which input to inject? Those representative of normal usage and those that makes no sense, you have to verify that the program has the functionalities that you expected and works, but you have either to try to construct particular commands that may be critical
- Example of fridge

*Vulnerabilities: important case studies*

- Adversarial != hostile
  - Hostile environment: inputs may be extreme and perhaps uncommon
  - Adversarial environment: inputs actively and carefully selected to provoke undesired behavior, the worst possible input, at the worst possible time

*NB non-SW engineers must come with hostile environments, SW engineers must cope with adversarial environments*

---

## Another interesting example

- Suppose you want to write a server that works according to this simple network protocol (prof invented):
  - Request message:
    - intereger someConst (predefined constant)
    - integer echo_lenght
    - byte array with echo_lenght elements
    - byte array with 2048 elements
      - Field 3 cannot be longer than 1024
  - Response message:respond with the byte array in field 3 of request (like an echo/ping)

```
#define LEN 4096
char respBuf[4096], reqBuf[4096];
...
// create and connect socket s
// receive request and store it in reqBuf (no overflow)
...
int *ptr = &reqBuf;
if (ptr[0] == someConst) {
   int len = ptr[1];
   memcpy(respBuf, (char*)ptr[2], len);   // No overflow
   // send response from respBuf
   ...
```

-Implementation outline: you define a constant LEN greater than the size of the request, create an array for storing the responses and one for the requests that you will receive
-Create a socket
-Receive a request and store it in the reqBuff (suppose that you are using safe memory libraries so even if the request is bigger than the buffer, the array will read at most its size, so there is no overflow)
-For constructing the response: pointer that point to the starting address of the received message, you make sure that the first integer received is the aspected one
-If it is, you check how long is the echo_lenght

  - You copy the third field in the response buffer according to the length (there is no overflow because if the request is all in the buffer, the response has also to fit in the other buffer with the same size)
  - You send back the response
- This is the same logic that led to a catastrophe: Heartbleed bug
  - The Heartbleed but allows anyone on the internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the service and users to impersonate services and users.
  - We ave tested some of our own services from attacker's prospective. We attacked ourselves form outside, without leaving a trace. Without using any privileged information or credentials we were able steal form ourselves the secret keys used fro our certificates, user names and passwords, instant messages, emails and business critical documents and communications
- There is a component of TSL called Heartbeat that works in this way: in a TLS connection, the client sends "are you still there? Please respond me with 'banana' which is 6 characters long" and the server responds "yes I'm here, you told me to send you 'banana'"
- The problem was that if the client sends the "are you there? Please respond me with 'banana' which is 100 chars long", the servers responds with 100 bytes, so banana+other informations (can be keys, passwords…what is below the information that the client requested)
- All because the server trusted the input

- You are sending back all bytes starting at `reqBuf+2*sizeof(int)`
- The amount of bytes is **an input data** (out of bound read!)

```
#define LEN 4096
char respBuf[4096], reqBuf[4096];
...
int *ptr = &reqBuf;
if (ptr[0] == someConst) {
   int len = ptr[1];
   memcpy(respBuf, (char*)ptr[2], len);   // No overflow
   ...
```

## Important lesson 2.0

- Tutto quello che è stato detto prima più altro:
- Memory unsafe programming languages are an unsafe practice
- There is never one reason:
- RFC 6520 was maybe not too explicit: if the payload_length of a received HeartbeatMessage is too large, the received HeartbeatMessage MUST be discarded silently, so the protocol is clear that in the second case (100 chars) the server does not have to respond, but the implementers of that protocol failed to follow that specification, because protocols are difficull
- Every protocol, at the and, has security considerations: the security considerations of RFC 5246 and RFC 6347 apply to this document. This document does not introduce only new security (there is no mention on the trust of the input, only on cryptographic problems)
- Never trust input data again
- Testing: this vulnerability (not the bug) could have been discovered automatically (OpenSSL has (at that time) only two full-time people to write, maintain, test and review 500000 lines of business critical code (2014))
- Il problema è anche legato al fatto che è tutto stato scritto in C, quindi non memory safe, fosse stato scritto in Java, l'input sbagliato non avrebbe permesso di andare ad accedere a parti di memoria sottostanti a quelle dedicate al programma