No VPN Needed? Cryptographic Attacks Against the OPC UA Protocol

By: Tom Tervoort, Bureau Veritas Cybersecurity

Summary

OPC UA is a standardized communication protocol that is widely used in the areas of industrial automation and IoT. It is used within and between OT networks, but also as a bridge between IT and OT environments or to connect field systems with the cloud. Traditionally, VPN tunnels are used to secure connections between OT trust zones (especially when they cross the internet), but this is often considered not to be neccessary when using OPC UA because the protocol offers its own cryptographic authentication and transport security layer.

This makes OPC UA a valuable target for attackers, because if they could hijack a (potentially internet-exposed) OPC UA server they might be able to wreak havoc on whatever industrial systems are controlled by it. Therefore, I decided to take a look at the cryptography used by the protocol, and whether any protocol-level flaws could be used to compromise implementations.

As a result, I managed to identify two crypto flaws which I could turn into practical authentication bypass attacks that worked against various implementations and configurations. These attacks involve signing oracles, signature spoofing padding oracles and turning "RSA-ECB" into a timing side channel amplifier that makes an otherwise theoretical flaw easily exploitable.

Background

In the world of OT (Operational Technology; i.e. computers that monitor or affect physical processes) a plethora of (often proprietary) communication protocols exists for systems to talk to each other. Like the OT systems that use them, many of these protocols lack security features: encryption is rare and if any credential is needed it is more often than not a default password you can find in the manual. That's why segmenting these types of networks, both on a network-level and physically, is so important. If you wouldn't Industroyer-style malware could hop over from the IT network after an employee gets phished, and one definitely doesn't want to expose these types of systems to the internet.

In contrast to these traditional protocols, OPC UA (Open Platform Communications Unified Architecture) provides an open and widely implemented standard that also offers security features. This makes it a popular protocol for communications that need to happen across security boundaries. OPC UA is for example used to connect PLC's in remote power substations, offshore windmills or gas pipelines with SCADA systems in central control rooms or data aggregators running on a cloud platform. All without the hassle of setting up IPSec tunnels.

Because of the way OPC UA links can cross important security boundaries, the protocol and its implementations are an attractive target for attackers. Therefore, security research is very much encouraged by the OPC Foundation: OPC UA was featured at Pwn2Own ISC and implementations have been the subject of extensive fuzzing and security research; in particularly by researchers from Claroty who have identified various vulnerabilities[1].

One aspect that hadn't received that much attention yet, however, is the bespoke cryptographic protocol OPC UA uses for authentication and transport security. Therefore I decided to take a look for potential cryptographic flaws in the OPC UA specification, and whether these could result in exploitable vulnerabilities against real implementations.

Cryptography in the OPC UA Protocol

OPC UA can operate as either a request/response based client-server protocol, or as a publish-subscribe system. The latter mechanism uses pretty straightforward authenticated cryptography based on a symmetric key negotiated via a client-server channel. The more complex cryptographic handshake and authentication mechanisms are all used over a client-server connection, so that is the part I focused my research on.

OPC UA supports both "application authentication" and "user authentication". Application authentication is based on X.509 certificates and is generally used to secure a connection between two automated systems. User authentication can use a variety of mechanisms like passwords, OAuth or also certificates; and is useful for authenticating human operators. An OPC Server administrator can choose whether to require either, neither or both authentication methods. Furthermore, they can also configure whether to mandate encryption or to allow connections only have integrity protection or no cryptographic protection at all.

Which authentication and encryption methods to use, if any, is determined via an "endpoint description". This description also contains the server certificate, which clients are supposed to validate according to a preconfigured PKI or allowlist. A server can have one or more endpoints with different security properties. An endpoint description can be preconfigured at the client-side, or be dynamically discovered via a "discovery server" or by querying the server itself (over an initially unencrypted and unauthenticated connection).

Once a client has chosen an endpoint description to use, it will exhange `Hello` and `Ack` messages with the server to set up connection parameters like message size limits. Then, a "secure channel" is established as follows:

[1] https://claroty.com/team82/research/white-papers/exploring-the-opc-attack-surface

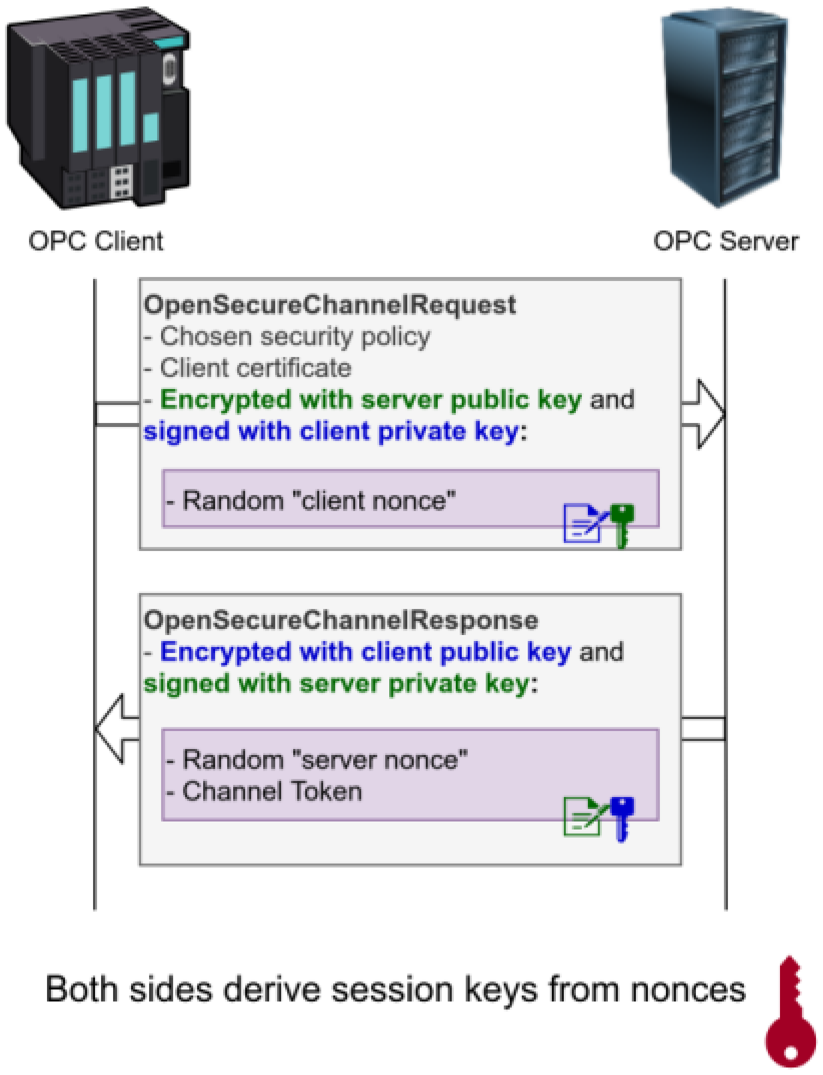

OPC Client
OPC Server
OpenSecureChannelRequest
- Chosen security policy
- Client certificate
- Encrypted with server public key and signed with client private key:
- Random "client nonce"
OpenSecureChannelResponse
- Encrypted with client public key and signed with server private key:
- Random "server nonce"
- Channel Token
Both sides derive session keys from nonces

Figure 1: Secure channel handshake.

1. The client will send an `OpenSecureChannel` message. If the endpoint description requires application authentication, it will transmit its own X.509 certificate and sign and encrypt the message with its private key.
2. The server validates the certificate, and then decrypts and validates the message. If succesful, it will extract a secret nonce from the message and send a reply with another nonce. This reply is signed and encrypted with the server's private key, matching its certificate.

If encryption or authentication is used, both sides will derive a shared secret from both nonces; and use it to symmetrically encrypt or authenticate subsequent messages. Mutual authentication is now performed via two exchanges within this channel:

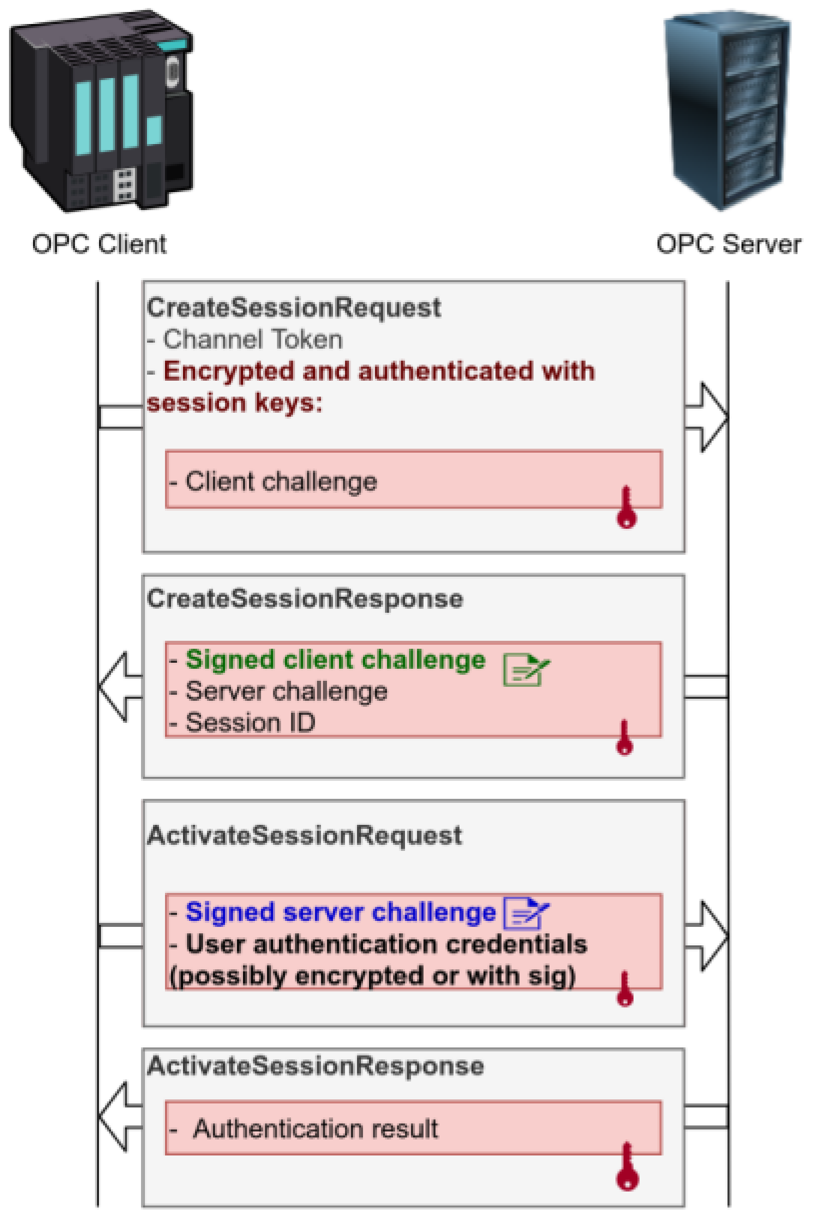


Figure 2: Creating an authenticated session within a secure channel. While the OPC specification uses the term "nonces" for the random values that are exchanged here as well, I use the term "challenge" in this diagram to prevent confusion with the nonces in the channel handshake.

3. The client sends a `CreateSession` request, containing a random challenge value (called `clientNonce` by the specification, but I will use the term `client challenge` here to distinguish it from the channel handshake nonces) and a copy of its certificate. The certificate must be identical to the one previously sent with the `OpenSecureChannel` request.

4. The server will respond with a cryptographic signature over the concatenation of `client certificate + client nonce` to authenticate itself. It will also send another challenge that will act as an authentication challenge for the client side, and a copy of its endpoint description.
5. The client checks server endpoints (to detect tampering during the initial unencrypted fetch) and validates the server signature. It will then send an `ActivateSession` request with a signature over `server certificate + server challenge` using its own key.
6. If user authentication is needed, the user's credential is added to the `ActivateSession` request as well. If certificate-based user authentication is in use, an additional signature over `server certificate + server challenge` is added with this certificate's key.
7. The server validates the client signature and user credentials, if any, and responds with an authentication token that the client will use to authorize subsequent requests.

Most implementations only support RSA for signing and asymmetric encryption. Unconventionally, no hybrid encryption is employed on the `OpenSecureChannel` messages. Instead the whole message is encrypted directly using RSA with (depending on the endpoint description) either PKCS#1 or OAEP padding. If the message does not fit in an RSA plaintext (which is usually the case because it also includes a signature), it is split into blocks that are encrypted individually and then concatenated, similarly to the ECB mode of operation for block ciphers.

Key derivation uses a PRF construction based on HMAC. When encryption is negotiated, a combination of HMAC and AES-CBC (in a bit of an odd "pad-then-MAC-then-encrypt" construction) is used for authenticated encryption of `CreateSession` and subsequent messages.

A pretty obvious downgrade vulnerability exists when a machine-in-the-middle were to alter the insecurely fetched server endpoint descriptions to make it appear as if the server only supports unencrypted and unauthenticated connections. However, it appears most client software I tested requires it to be explicitly configured whether to use encryption or not. Once encryption was enabled, all clients I tested appeared to reject this type of downgrade attack. Downgrading from encrypted to authentication-only connections also did not appear to be possible due to the endpoint validation process in step 5.

I did however identify two other handshake vulnerabilities that were exploitable in practice. These are detailed below.

Attack 1: Authentication Bypass via a Signing Oracle in OPC over HTTPS

Vulnerability

One flaw I identified was that the signatures used within the server's `CreateSession` response and the client's `ActivateSession` request do not include any context information about what type of signature it is. Client signatures, server signatures and user signatures all use the exact same `peer certificate + peer challenge` format. Furthermore, the party that signs first is the server, opening up a theoretical "signing oracle" vulnerability where the client tricks the server into signing some other server's certificate and challenge. The client can then copy this server signature into its own client signature field and impersonate a server.

However, before an attacker is in the position to send a `CreateSession` message they will first need to be able to bypass the `OpenSecureChannel` handshake, which already requires them to implicitly prove ownership of their private key by signing a request and decrypting a response. Since the certificate used in the `OpenSecureChannel` must be identical to the one in the `CreateSession` request, they should not be able to impersonate anyone else anymore.

Due to this, the use of signatures within the create and activate session messages appears to be redundant. At least, that is the case when OPC UA is used with its most common transport layer: plain TCP. But the standard also describes an alternative transport method where OPC UA messages are packed into HTTPS requests and responses. Because these messages are already protected by TLS, the HTTPS variant of the protocol completely skips the `OpenSecureChannel` exchange, and instead the `CreateSession` and `ActivateSession` messages are relied upon to authenticate the client.

This causes the vulnerability to be exploitable, but only over the HTTPS transport. I identified two variations of this exploit: a "relay attack" and a "reflection attack".

Relay attack

When an attacker can reach two OPC UA servers that both support the HTTPS transport and trust each other's certificates, they can use the signing oracle vulnerability to impersonate one server towards the other. This attack works as follows:

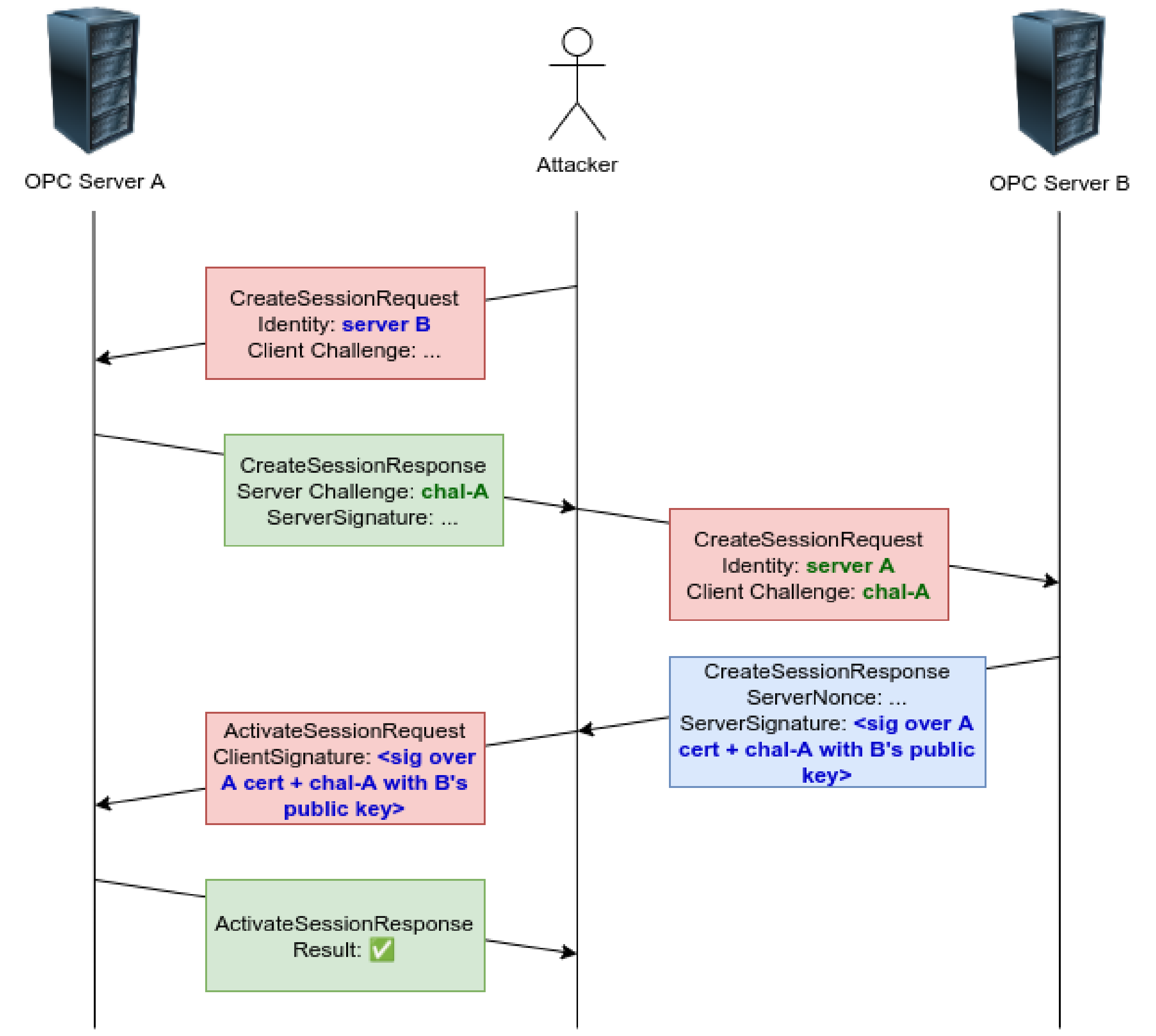


Figure 3: Relay attack illustration.

1. Via the endpoint discovery mechanism, the attacker fetches certificates of two servers A and B; name those `cert-A` and `cert-B`.
2. The attacker sends a `CreateSession` request to server A, containing `cert-B` and some arbitrary challenge.
3. The attacker receives a response from server A, asking to sign their server challenge `challenge-A`.
4. The attacker sends a `CreateSession` request to server B, with `cert-A` and `challenge-A`.
5. B responds with a signature over `cert-A + challenge-A`, with B's private key.
6. The attacker sends an `ActivateSession` request to server A, relaying the signature received

from B.
7. A successfully validates the signature with B’s public key, and grants an authentication token to the attacker with the identity of server B.

Reflection attack

Often, an OPC UA server will also trust a certificate that is completely identical to the one the server uses itself. This appeared to be the default behavior of some implementations, and will also happen when the same root CA is used to validate both client and server certificates. When this is the case, it is also possible to execute a “reflection attack”, where two interleaving handshakes are done against the same server. The attack is effectively the same as a relay attack, except that server A and server B are the same:

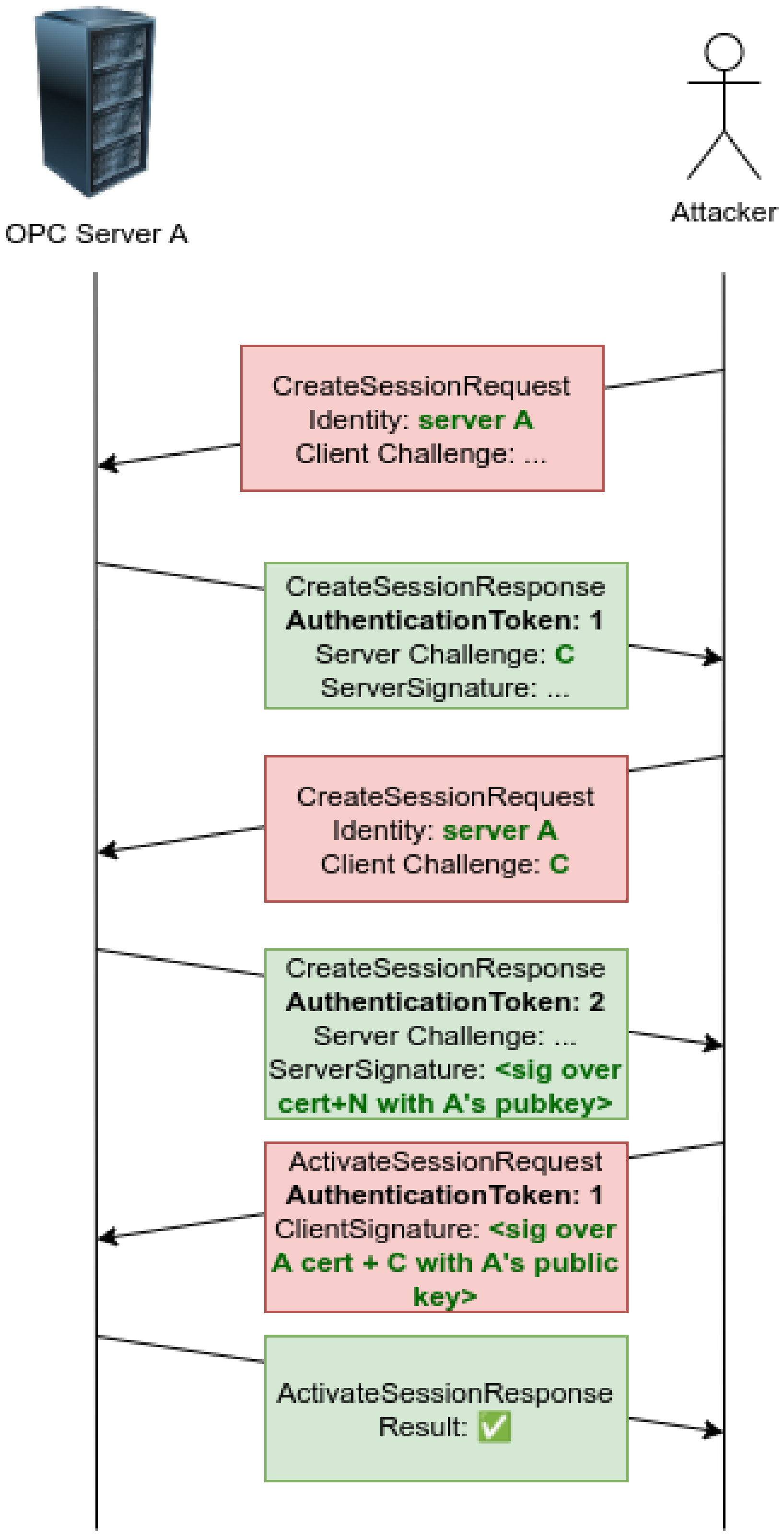


Figure 4: Reflection attack illustration.

Impact

While the HTTPS transport is rarely used in practice, I found two implementations (including the UA-.NETStandard reference implementation) that expose an OPC-over-HTTPS interface by default. If the HTTPS port were reachable, a reflect attack could be used against both implementations to achieve a full client authentication bypass.

This attack bypasses application authentication but not necessarily user authentication. The latter is probably not required if an OPC server is intended for system-to-system communication. However, if the server does need a user password then the attacker will still need to guess it (e.g. via dictionary attacks, phishing or credential stuffing) in order to proceed. If user authentication based on certificates is used, a reflected or relayed signature may also be used within the user certificate field, which will work whenever the same PKI is used for user and application certificates.

Attack 2: Authentication Bypass or Decryption via an RSA Padding Oracle

Vulnerability

Part of an endpoint description is a "security policy", which is an identifier corresponding to a "cipher suite" of cryptographic primitives. Most commonly implemented policies use RSA with OAEP padding for asymmetric encryption, but this is not the case for the officially deprecated but still widely supported policy `Basic128Rsa15`. This policy uses SHA-1 for hashing, AES-128 for symmetric encryption and RSA with the PKCS#1 padding scheme for signing and asymmetric encryption.

While PKCS#1 is generally fine for signatures, using it for encryption has been shown to be insecure by Daniel Bleichenbacher back in 1998[2]. If an attacker can find a "padding oracle" (some system behavior that tells them whether a specifically chosen RSA ciphertext is correctly padded or not after decryption) they can use that to both decrypt ciphertexts or forge signatures with that system's private RSA key.

Due to its use of the insecure SHA-1 hash function, the OPC Foundation had already officially deprecated the `Basic128Rsa15` and stated that implementations should disable it by default and discourage its use in their documentation. However, there may still be plenty of systems that were configured to allow this policy before the deprecation, and would retain this configuration even as the software got updated.

Furthermore, it should be taken into account that a PKCS#1 padding oracle can also be used to break encryption or signatures using different (otherwise secure) padding schemes like OAEP or PSS, as long

[2]Bleichenbacher, Daniel. "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1." *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18*. Springer Berlin Heidelberg, 1998. https://archiv.infsec.ethz.ch/education/fs08/secsem/bleichenbacher98.pdf

as the same RSA key is reused. OPC UA does not prohibit using the same RSA key with different padding schemes, and this is exactly what all implementations I tested appeared to be doing. This means that when a server explicitly disables `Basic128Rsa15` itself but still trusts some other legacy system that still allows it, then that legacy system can be abused to attack the securely configured server.

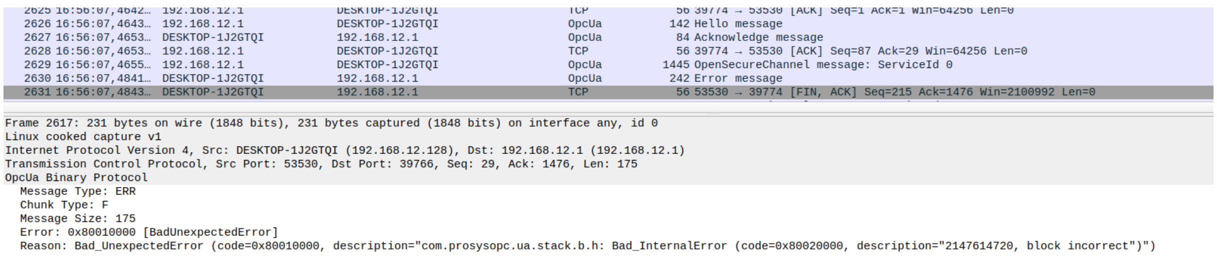
I also found two implementations that attempted to decrypt a PKCS#1 ciphertext before actually validating if this algorithm was allowed to be used, which was enough to make the padding oracle exploit work in a default configuration where `Basic128Rsa15` was explicitly disabled.

In order to exploit this, the first step is to find a padding oracle: some piece of functionality that will attempt to decrypt attacker-chosen ciphertexts and somehow reveals whether the result had correct padding or not. I found three types of padding oracle, which are explained below.

Error-based padding oracle in OpenSecureChannel

When an OPC UA server receives an `OpenSecureChannel` request it is supposed to first decrypt the request payload and then verify the signature on it. If the padding is incorrect this would cause decryption to fail, whereas a random message with correct padding would cause signature verification to fail. If the returned error messages for decryption errors and validation errors are different, that means a padding oracle is present.

This turned out to be the case for one implementation: while it would return the same error code for both errors, the message had an additional `Reason` field that would contain different error messages at each situation.



```
  2625 16:56:07,4642…  192.168.12.1      DESKTOP-1J2GTQI    TCP         56 39774 → 53530 [ACK] Seq=1 Ack=1 Win=64256 Len=0
  2626 16:56:07,4643…  192.168.12.1      DESKTOP-1J2GTQI    OpcUa      142 Hello message
  2627 16:56:07,4653…  DESKTOP-1J2GTQI   192.168.12.1       OpcUa       84 Acknowledge message
  2628 16:56:07,4653…  192.168.12.1      DESKTOP-1J2GTQI    TCP         56 39774 → 53530 [ACK] Seq=87 Ack=29 Win=64256 Len=0
  2629 16:56:07,4655…  192.168.12.1      DESKTOP-1J2GTQI    OpcUa     1445 OpenSecureChannel message: ServiceId 0
  2630 16:56:07,4841…  DESKTOP-1J2GTQI   192.168.12.1       OpcUa      242 Error message
  2631 16:56:07,4843…  DESKTOP-1J2GTQI   192.168.12.1       TCP         56 53530 → 39774 [FIN, ACK] Seq=215 Ack=1476 Win=2100992 Len=0

Frame 2617: 231 bytes on wire (1848 bits), 231 bytes captured (1848 bits) on interface any, id 0
Linux cooked capture v1
Internet Protocol Version 4, Src: DESKTOP-1J2GTQI (192.168.12.128), Dst: 192.168.12.1 (192.168.12.1)
Transmission Control Protocol, Src Port: 53530, Dst Port: 39766, Seq: 29, Ack: 1476, Len: 175
OpcUa Binary Protocol
  Message Type: ERR
  Chunk Type: F
  Message Size: 175
  Error: 0x80010000 [BadUnexpectedError]
  Reason: Bad_UnexpectedError (code=0x80010000, description="com.prosysopc.ua.stack.b.h: Bad_InternalError (code=0x80020000, description="2147614720, block incorrect")")
```

Figure 5: Example of a distinct error message (of the Prosys implementation) when an RSA plaintext has invalid padding.

Error-based padding oracle in user authentication

The other implementations I tested did not have observable differences in their `OpenSecureChannel` error responses. Three of them did however have an error-based padding oracle in their password decryption functionality, that is used during password-based user authentication. However, an attacker can only reach the point where they can submit an encrypted user password when a server does not enforce client authentication. So these padding oracles would be mostly useless for an authentication bypass attack, but could still be used to decrypt sniffed passwords.

Amplified timing-based padding oracle

Even when a server does not show different error messages, it may still be possible to tell the two errors apart based on the time it takes for a server to process the message. In the case of OPC UA, the server will attempt to validate an RSA signature when decryption succeeds, but may terminate right away when it fails. This opens up a theoretical side-channel attack where response timing is used to tell the difference between the two cases.

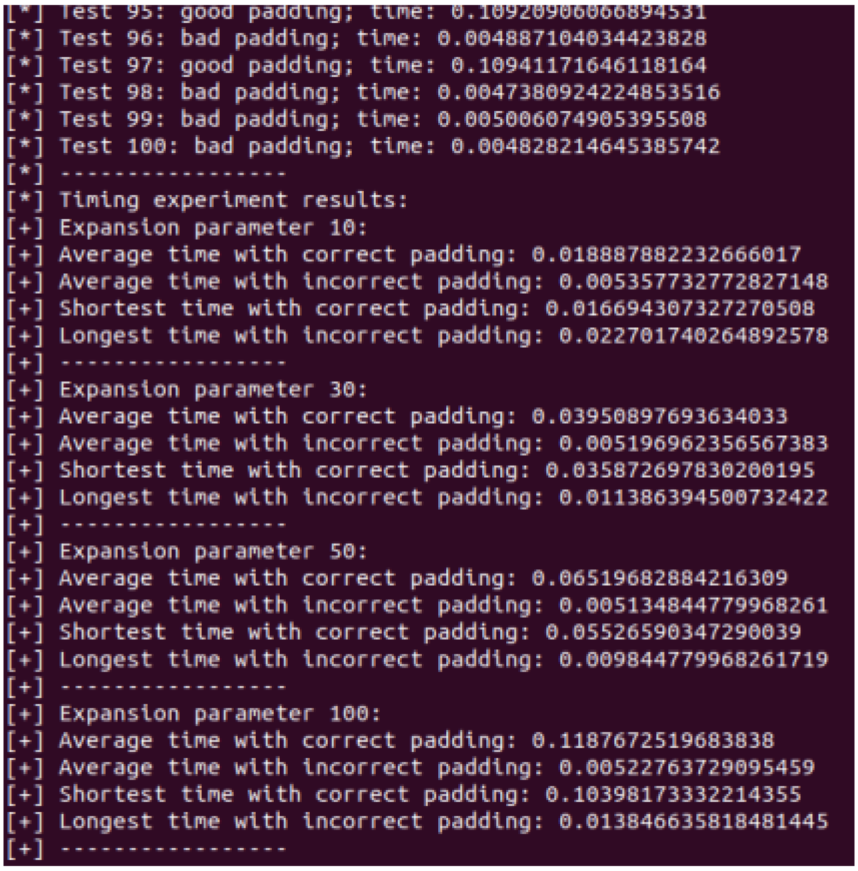
Because the timing difference of these two cases is pretty small, carrying out such a timing attack over the network would likely be very difficult and require many measurements to filter out the noise caused by network latency and such.

However, it turns out that there was a way to amplify the timing difference between the two kinds of errors by taking advantage of OPC UA's non-standard way of decrypting messages that are too long to fit within a single RSA ciphertext. For long messages, the input is treated as a series of concatenated ciphertexts that are decrypted and unpadded one by one. Finally, all resulting plaintexts are concatenated together and signature validation starts.

RSA decryption is an expensive operation. If you were to submit a concatenation of 100 correctly padded ciphertexts the server would have to do 100 RSA decryptions before it can even begin signature validation. However, if you submit 100 incorrectly padded ciphertexts then the server would only do a single decryption, which will fail and cause it to respond with an error code right away.

To take advantage of this during a padding oracle attack, you can simply repeat each guessed ciphertext a lot of times within the `OpenSecureChannel` message. When the server responds right away you treat this guess as having wrong padding. When the server has a big noticeable delay, you retry the same message a few times to see if this delay happens consistently. If so, you can conclude that guess must have had valid padding.

I made my attack tool run some simple timing tests where both correctly and incorrectly padded messages were respectively repeated 10, 30, 50 and 100 times. In four implementations, I found huge timing differences that could be measured in tens of seconds or even in full seconds; allowing me to carry out succesful timing-based padding oracle attacks where the simple retry strategy was sufficient to address noise.



```
[*] Test 95: good padding; time: 0.1092090066894531
[*] Test 96: bad padding; time: 0.004887104034423828
[*] Test 97: good padding; time: 0.1094117164618164
[*] Test 98: bad padding; time: 0.0047380924224853516
[*] Test 99: bad padding; time: 0.005006074905395508
[*] Test 100: bad padding; time: 0.004828214645385742
[*] ----------------
[*] Timing experiment results:
[+] Expansion parameter 10:
[+] Average time with correct padding: 0.01888788223266017
[+] Average time with incorrect padding: 0.005357732772827148
[+] Shortest time with correct padding: 0.016694307327270508
[+] Longest time with incorrect padding: 0.022701740264892578
[+] ----------------
[+] Expansion parameter 30:
[+] Average time with correct padding: 0.03950897693634033
[+] Average time with incorrect padding: 0.005196962356567383
[+] Shortest time with correct padding: 0.035872697830200195
[+] Longest time with incorrect padding: 0.011386394500732422
[+] ----------------
[+] Expansion parameter 50:
[+] Average time with correct padding: 0.06519682884216309
[+] Average time with incorrect padding: 0.005134844779968261
[+] Shortest time with correct padding: 0.05526590347290039
[+] Longest time with incorrect padding: 0.009844779968261719
[+] ----------------
[+] Expansion parameter 100:
[+] Average time with correct padding: 0.1187672519683838
[+] Average time with incorrect padding: 0.00522763729095459
[+] Shortest time with correct padding: 0.10398173332214355
[+] Longest time with incorrect padding: 0.013846635818481445
[+] ----------------
```

Figure 6: Example of timing tests carried out by the attack tool.

Feasibility

Bleichenbacher's attack is also sometimes called the "million messages" attack because it may require up to a million padding oracle interactions before a full RSA decryption is complete. How fast this is highly depends on the speed of the target and the connection as well as the general reliability of the oracle. Still, sending (and waiting for the processing of) a million messages is not that expensive. Depending on the implementation, the time it took to execute a full attack in my lab setup ranged from 10 minutes to around three hours, even though I only implemented a straightforward and non-parallelized implementation of the 1998 without any of the optimizations researchers have proposed since. In many attack scenarios, waiting a few hours to achieve an authentication bypass is not that great of an ask.

Interestingly, the timing-based oracles were not that much slower than the error-based once. This is the case because only about one in a thousand queries result in a positive result (i.e. indicating that padding is correct). This case is also the "slow" case of the timing attack, whereas the server would respond quickly for the great majority of cases when the padding was incorrect. In fact, the fastest attack (taking 10 minutes) was carried out against an implementation that only exposed a timing-based oracle.

Turning a padding oracle into an authentication bypass

So assuming you can carry out Bleichenbacher's padding oracle attack against some server, how do you turn that into a practical attack against OPC UA? What the attack basically allows you to do is to make the server reveal the result of an RSA decryption operation on a value that you choose. So if you would have passively sniffed an encrypted OPC UA connection, you would be able to decrypt the `OpenSecureChannel` exchange (given that the client can also act as a server and both sides are accessible to the attacker) and extract the secret nonces, allowing you to decrypt the rest of the traffic.

An authentication bypass is probably more interesting, though, but this requires signature spoofing. Bleichenbacher's attack can also also be used for that purpose, because after padding RSA signing is effectively the same as RSA decryption. So when you provide the padded hash of a message you would like to sign as a "ciphertext", then the associated "plaintext" will be a valid signature over the message. This allows the following attack to be put together:

1. Prepare an `OpenSecureChannel` message and use the padding oracle to get it signed.
2. Encrypt the signed message with the server's public key (since it's public key encryption, this step does not need the oracle) and use it to start a handshake with that server.
3. Receive the encrypted response and use the padding oracle again to decrypt it, extract the nonce, and compute the session key.
4. Now you have gotten past the `OpenSecureChannel` step of the handshake, and can use attack 1 (a reflect or relay attack) to bypass client authentication. Alternatively, if that's not possible for some reason, you can also invoke the padding oracle again to forge the neccessary client signature.

What makes the attack a bit tricky is that step 3 must be carried while keeping an active connection open, because otherwise the server would forget about the session and discard the exchanged key material. I found this was pretty easy to achieve against most implementations by simply sending regular TCP keep-alive messages, even when this needed to be done for several hours. Occasionally, this would fail and steps 2 and 3 would have to be repeated. Conveniently, the result of step 1 is reusable across sessions, so only the second of the two padding oracle attacks needs to be repeated when the connection is lost.

Impact

A server with a vulnerable implementation or configuration becomes vulnerable to the same types of reflect and relay attacks as described under attack 1, except that they no longer need to provide an HTTPS endpoint. On the attacker's side, they have the constraint that the attack will take more time (but probably not more than a few hours per server) and will produce more obvious indicators of compromise in logs.

Unlike attack 1, the padding oracle can also be used to subvert transport security of existing connections; enabling decryption of passively intercepted traffic and (in theory) active machine-in-the-middle attacks.

Attack tool

I created a Python tool named `opcattack` that I have so far succesfully used to create PoC's against five different OPC UA implementations. The tool can check for potentially vulnerable servers and automates several varieties of both attacks and can be downloaded at https://github.com/SecuraBV/opcattack.

It currently supports reflection and relay attacks over HTTPS; reflection attacks over TCP with a padding oracle; and general message decryption or signature forging with a padding oracle. Each padding oracle attack supports both error-based oracles (capable of distinguishing errors generated by at least three different implementations) and implementation-agnostic timing-based oracles (along with a testing mode that helps picking appropriate attack parameters).

The tool has been tested against the seven implementations listed under "Affected implementations". I expect it to have decent compatibility with other standards-compliant OPC UA implementations, although it may not be able to detect error-based padding oracles in them.

Affected implementations

I tested both attacks against seven different OPC UA servers, with the following results:

Implementation	Tested version	Vulnerable to attack 1	Vulnerable to attack 2
dataFEED edgeConnector	2024.01	no	no**
Ignition	8.1.38	no	yes
KEPServerEX	6.15.154.0	no	yes*
open62541	1.4	no	yes

Implementation	Tested version	Vulnerable to attack 1	Vulnerable to attack 2
Prosys OPC UA Simulation Server	5.4.6-180	yes	yes
UA-.NETStandard Reference Server	1.4.372-preview	yes	yes*
Unified Automation C++ Demo Server	1.8.2.624	no	no**

* Only vulnerable when Basic128Rsa15 is enabled, which wasn't the case by default in a fresh installation.
** I couldn't find an error-based or amplified timing oracle, but can't rule out a more subtle timing-based oracle attack may be possible.

Because the issues are related to protocol flaws more implementations may be affected. I have for example received confirmation that a variety of Siemens products (including WinCC Unified)[3] were vulnerable to both attacks.

Mitigations

The protocol vulnerabilities have been responsibly disclosed to the OPC Foundation, who have issued CVE-2024-42512 and CVE-2024-42513 and coordinates disclosure to OPC UA implementors, who issued advisories and patches for their products if applicable.

If you use OPC UA and rely on client authentication, you can refer to your vendors documentation or release notes to check whether these CVE's have been addressed. When it's unclear if an implementation is affected, my opcattack may help checking for the vulnerabilities.

Patched or not, it is probably prudent to disable the HTTPS transport and Basic128Rsa15 security policy whenever these are not strictly neccessary. If the product allows configuring a separation between trusted client-side and server-side certificates, I also recommend making sure that there is no overlap between those; and that certificates are not reused for both client and server roles.

Conclusion and future research

So should everyone wrap all their OPC UA links into VPN tunnels now? Probably not, as in most cases both attack vectors can be mitigated by patching the servers, without breaking compatibility with unpatched clients. Of course if an internet-exposed server in question is highly sensitive, or if you are

[3] https://www.cisa.gov/news-events/ics-advisories/icsa-25-072-09

in a situation where you need to expose a server that can not be easily patched, using either a VPN tunnel or at least some IP allowlisting would be prudent anyway.

However, I would like to argue that we should probably count on more (exploitable) cryptographic vulnerabilities to be discovered in the future. Considering the protocol has been first published in 2006, it is not surprising that it does not adhere to modern best practices for cryptographic protocol design. I highly doubt my research was anywhere near exhaustive, and considering some of the exotic cryptographic constructions that are being used within OPC UA I think there is plenty of interesting material left for vulnerability researchers.