Go x/crypto/ssh Cryptographic Implementation Review

Teleport

August 6, 2026

Version 1.0

Prepared for:
Geomys and the Go crypto Maintainers,
on behalf of Teleport

Prepared by:
Kevin Henry
Marie-Sarah Lacharité
Thomas Pornin
Javed Samuel
Eric Schorn

1 Executive Summary

This report presents the findings of the Go x/crypto/ssh Cryptographic Implementation Review conducted on behalf of Teleport. The assessment was conducted during December 2025 and January 2026 with an effort of 25 person-days. The review was also supported by Geomys, who provided technical support, disclosure coordination and fixes to the findings identified in this report.

The review targeted the SSH protocol implementation included in Go's standard library supplementary cryptographic package (`x/crypto/ssh`, commit `f4602e4`). Both client and server sides of the protocol are supported.

A retest was performed in April 2026, which found that all findings had been fixed by Geomys. Follow-up testing confirmed the fixes were merged upstream during May and June.

Overview

The assessment uncovered several findings, with the highest severity items including:

- Maliciously crafted (invalid) public keys may induce peers to spend inordinate amounts of CPU power during validation, in some cases up to several minutes worth of computational cost for a single connection (finding "Abnormal RSA and DSA Public Keys Incur High Cost for Signature Verification").
- When multi-factor authentication is performed, specific permissions and restrictions associated with one factor are silently dropped, potentially allowing a much wider access than expected (finding "SSH Multi-Factor Authentication Drops Public-Key Permissions").

The following table breaks down the issues which were identified by component and severity of risk:

Component	Critical	High	Medium	Low	Informational	Total
General	0	2	6	5	2	**15**
Total	**0**	**2**	**6**	**5**	**2**	**15**

A total of nine (9) CVEs were created to track issues identified in this report.

Limitations are described in the Caveats section, and additional commentary is provided in the Engagement Notes appendix.

Assessment Summary

Teleport/Geomys promptly confirmed and remediated all findings identified in a manner consistent with the recommendations made after the initial assessment. Upon retesting, all findings were deemed to be *Fixed*. Geomys also coordinated the CVE disclosure process.

Strategic Recommendations

NCC Group recommends that more thorough data validation is applied on received values, not only from on-the-wire packets but also from API calls on the library, to avoid or mitigate denial-of-service attacks in some usage contexts.

NCC Group found that the code was generally well-structured and documented and recommends ensuring that these high standards are maintained going forward. In particular, predictable modifications to the protocol to include more upcoming post-quantum cryptographic algorithms will require some corresponding support code, which should carefully follow the existing code framework.

2 Project Information

Assessment Name	x/crypto/ssh
Environment	Local
Assessment Dates	2025-12-09 to 2026-04-24
Number of Consultants	4
Level of Effort	25 Person-days
Type	Cryptographic Review
Method	Code-assisted

Scope

crypto/ssh/

Targets

The SSH implementation located in the *crypto/ssh/* directory of the *x/crypto* Go package, representing approximately 22k lines of code. At the beginning of the engagement, the most recent commit in the source code repository was `f4602e4...` (dated December 2nd, 2025); the review was performed on the source code corresponding to that specific commit.

https://cs.opensource.google/go/x/crypto/+/f4602e40409257658159002a9af6aedb875949fb:ssh/

Standalone patches and proposed updates

Targets

A small number of changes/updates were reviewed. The source code modifications implied by these patches were almost entirely contained within the ssh-agent support code.

https://github.com/drakkan/agent/tree/16b0e02356b192cecb6781bd91b2bd6439c8ab77

Caveats

The SSH implementation is a library that can be configured in various ways by the application that uses it, not only in the choice of supported cryptographic algorithms, but also by using application-provided callbacks for some features such as validating host and user public keys, as well as integrating the protocol within an arbitrary transport layer. NCC Group's evaluation assumed that applications use the library correctly (as per the provided API documentation), and the possible impact and severity of findings were estimated by following NCC Group's notion of what a generic application might realistically do with the library.

3 Findings Breakdown

The graphic below is a breakdown of the findings by the **Overall Risk** of each issue:

Finding Breakdown

Critical issues	0	
High issues	2	
Medium issues	6	
Low issues	5	
Informational issues	2	
Total issues	**15**	

The graphic below is a breakdown of the findings by the **Category** of each issue:

Category Breakdown

Configuration	2	
Cryptography	4	
Data Validation	5	
Denial of Service	4	

Critical High Medium Low Informational

4 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
SSH Multi-Factor Authentication Drops Public-Key Permissions	Fixed	LRN	High
Abnormal RSA and DSA Public Keys Incur High Cost for Signature Verification	Fixed	JVF	High
SSH Agent Client Drops Constraint Extensions	Fixed	ET6	Medium
Nil Dereference Panic on `c.IsUserAuthority` and `c.IsHostAuthority`	Fixed	HHY	Medium
Hang on Unsolicited Global Request Replies	Fixed	WNA	Medium
Maliciously Crafted Private Keys Can Trigger Panics or Very High CPU Usage	Fixed	GDJ	Medium
Insufficient Validation of `ed25519` Private Key	Fixed	BWW	Medium
Insufficient Validation of Userauth Messages Before Signing	Fixed	MN2	Medium
Large Channel Writes May Stall Forever	Fixed	NTF	Low
`SK*` Signatures Ignore User Presence Flag	Fixed	C6P	Low
Incorrect Handling of Usernames in Multi-Hop Constraint Validation with Binding	Fixed	JGV	Low
Insufficient Validation When Parsing Known Hosts	Fixed	CAL	Low
Suboptimal Dynamic Diffie-Hellman Modulus Selection	Fixed	R6B	Low
SSH Agent Keyring Ignores `Confirm-Before-Use` Constraint	Fixed	BCX	Info
Undetected Public Key Inconsistencies	Fixed	K3R	Info

5 Finding Details

High

SSH Multi-Factor Authentication Drops Public-Key Permissions

Overall Risk	High	**Finding ID**	NCC-E029268-LRN
Impact	High	**Category**	Cryptography
Exploitability	Medium	**Status**	Fixed

Impact

Multi-factor flows utilizing `PartialSuccessError` discard the `Permissions` struct returned by earlier factors, so certificate critical options (e.g., `force-command`, `permit-agent-forwarding`, or custom extensions) never reach the application once a second factor succeeds. A user who is meant to be confined to those restrictions can therefore obtain an unrestricted session simply by completing the additional factor.

Description

The `NewServerConn()` function implemented in the *ssh/server.go* source file starts a new SSH server given an underlying transport and `ServerConfig`, executes the `serverHandshake()` function which then invokes the `serverAuthenticate()` function to execute the configured authentication callbacks. Callbacks are expected to return a `Permissions` object whose `CriticalOptions` and `Extensions` fields may carry certificate restrictions such as `force-command`, `permit-agent-forwarding`, or any custom metadata that the application layer enforces after authentication.

The `userAuthLoop` within the `serverAuthenticate()` function excerpted below effectively iterates through received multi-factor methods. However, the loop resets `perms` to `nil` on every pass and only keeps whatever the last successful method returned.

```
func (s *connection) serverAuthenticate(config *ServerConfig) (*Permissions, error) {
...
userAuthLoop:
  for {
    ...
    var userAuthReq userAuthRequestMsg
    if packet, err := s.transport.readPacket(); err != nil {
      if err == io.EOF {
        return nil, &ServerAuthError{Errors: authErrs}
      }
      return nil, err
    } else if err = Unmarshal(packet, &userAuthReq); err != nil {
      return nil, err
    }
    ...
    perms = nil
    authErr := ErrNoAuth

    switch userAuthReq.Method {
    case "none":
      ...
    case "password":
```

```
    ...
  case "keyboard-interactive":
    ...
  case "publickey":
    ...
    candidate, ok := cache.get(s.user, pubKeyData)
    if !ok {
      candidate.user = s.user
      candidate.pubKeyData = pubKeyData
      candidate.perms, candidate.result = authConfig.PublicKeyCallback(s, pubKey)
    ...
    if err := pubKey.Verify(signedData, sig); err != nil {
      return nil, err
    }

    authErr = candidate.result
    perms = candidate.perms
  ...
  default:
    authErr = fmt.Errorf("ssh: unknown method %q", userAuthReq.Method)
  }  // End of switch statement
  ...
  if partialSuccess, ok := authErr.(*PartialSuccessError); ok {
    // After a partial success error we don't allow changing the user
    // name and execute the NoClientAuthCallback.
    partialSuccessReturned = true

    // In case a partial success is returned, the server may send
    // a new set of authentication methods.
    authConfig = partialSuccess.Next

    // Reset pubkey cache, as the new PublicKeyCallback might
    // accept a different set of public keys.
    cache = pubKeyCache{}

    // Send back a partial success message to the user.
    failureMsg.PartialSuccess = true
  } else ...
}  [[[[// Loop continues (line 893)
```

Figure 1: Excerpt of `serverAuthenticate()` from ssh/server.go

When the `PublicKeyCallback()` function approves a key, it can legitimately return both a `Permissions` object and a `PartialSuccessError` to force a subsequent authentication method. The code above assigns `candidate.perms` to the loop-local `perms`, but as soon as the partial-success path runs it changes the callback set and immediately iterates again with `perms = nil`. There is no state that remembers the previously granted permissions, so by the time another method succeeds, the `perms` pointer ultimately handed back to the `NewServerConn()` function contains only the permissions from that last factor.

Consider the a certificate+OTP flow implemented by `CertChecker.Authenticate()`: when a user presents a certificate that contains `CriticalOptions["force-command"] = "logger --tag audited"`, the callback returns those permissions along with `PartialSuccessError{Next: ServerAuthCallbacks{PasswordCallback: otp}}` to demand a second factor. After the client answers the OTP challenge, the `serverAuthenticate()` function overwrites `perms` with the

password callback's return value (perhaps `nil`), `ServerConn.Permissions` is empty, and the application layer never sees the certificate restrictions. As a result, any remote user who possesses a restricted certificate plus the required second factor can log in and obtain an unrestricted SSH session, bypassing forced commands, enabling forwarding, or otherwise escalating account capabilities.

Recommendation

Preserve and merge the `Permissions` returned by earlier authentication steps whenever a `PartialSuccessError` is emitted.

Location

The `serverAuthenticate()` function implemented in *ssh/server.go*

Retest Results

2026-04-10 – Fixed

The project team reviewed *0002-ssh-enforce-nil-Permissions-when-returning-PartialSu.patch*, which updates `serverAuthenticate()` to return an error if permissions are not `nil` when `PartialSuccessError` is encountered. This check is stricter than the recommendation above, and therefore addresses the core issue in this finding (this method also avoids the potentially non-trivial question of whether merging should return the union or the intersection of the permissions associated with the authentication factors). Regression tests are included.

In summary, this finding is considered *Fixed*.

2026-06-02 – Fixed

The reviewed patch was merged in https://go-review.googlesource.com/c/crypto/+/781621, and publicly disclosed as CVE-2026-39828 and GO-2026-5014.

High

Abnormal RSA and DSA Public Keys Incur High Cost for Signature Verification

Overall Risk	High	**Finding ID**	NCC-E029268-JVF
Impact	High	**Category**	Denial of Service
Exploitability	Medium	**Status**	Fixed

Impact

An attacker can send malformed authentication packets that trigger a signature verification (RSA or DSA) that will consume large amounts of CPU on the peer before ultimately failing (up to 5 minutes worth of CPU time for a single attack instance). Applicability on either the client or the server depends on the configuration and application-provided callbacks.

Description

There are several contexts in which either the SSH client or the SSH server verifies cryptographic signatures against a public key sent along by the peer. Public keys are nominally validated to be correct (within the SSH security model), but the validation of the public key might happen only after the signature verification has taken place. In the case of the RSA and DSA algorithms, low-level public key and signature decoding functions verify only a few properties on the obtained values, and may accept maliciously crafted invalid values whose processing in a signature verification entails large CPU usage.

RSA: A RSA public key is represented as a couple of big integers ("mpint" in SSH terminology) for the modulus n and the public exponent e. The `parseRSA()` function (defined in *ssh/keys.go*) decodes the two mpint values and checks that the public exponent is odd and fits on 24 bits, but it performs no check on the modulus:

```
463     var w struct {
464         E    *big.Int
465         N    *big.Int
466         Rest []byte `ssh:"rest"`
467     }
468     if err := Unmarshal(in, &w); err != nil {
469         return nil, nil, err
470     }
471
472     if w.E.BitLen() > 24 {
473         return nil, nil, errors.New("ssh: exponent too large")
474     }
475     e := w.E.Int64()
476     if e < 3 || e&1 == 0 {
477         return nil, nil, errors.New("ssh: incorrect exponent")
478     }
479
480     var key rsa.PublicKey
481     key.E = int(e)
482     key.N = w.N
```

Figure 2: Excerpt from ssh/keys.go

Signature values nominally have the same size as the modulus (as a sequence of bytes) but some third-party SSH implementations have the habit of removing leading bytes of value

zero; the implementation supports such “shortened” signatures in the interest of interoperability (see *ssh/keys.go* lines 515 to 555), so that even if the modulus is very large, then the accompanying signature needs not take a commensurate size in the sent packet. The maximum allowed packet size is 262144 bytes (see `maxPacket` in *ssh/cipher.go*), so that an attacker may hope to send a crafted public key with a modulus of up to slightly above 2 million bits in size.

RSA verification is performed by Go’s standard library (`crypto/rsa.VerifyPKCS1v15()`); it uses a modular exponentiation, which resolves to a sequence of modular multiplications. The number of such multiplications is linear in the public exponent size, which is here limited to 24 bits; however, each modular multiplication has a cost which is quadratic in the modulus size, so that large moduli imply a high cost. On an Intel x86 CPU (Intel i5-8259U at 2.3 GHz, Coffee Lake core, no frequency scaling, 64-bit mode, Linux operating system, Go 1.25.5), the verification keeps one CPU core at 100% usage for up to **5 minutes** when the modulus size is slightly above 2 million bits[1].

DSA: A DSA public key includes a modulus p, a secondary modulus q (nominally a divisor of $p - 1$), a generator g and a public value y (p, q and g being the “DSA parameters”, potentially shared between several key pairs). The `parseDSA()` function decodes these values from mpint representations; the only check which is performed is that p is a 1024-bit integer:

```
569 func checkDSAParams(param *dsa.Parameters) error {
570     // SSH specifies FIPS 186-2, which only provided a single size
571     // (1024 bits) DSA key. FIPS 186-3 allows for larger key
572     // sizes, which would confuse SSH.
573     if l := param.P.BitLen(); l != 1024 {
574         return fmt.Errorf("ssh: unsupported DSA key size %d", l)
575     }
576
577     return nil
578 }
```

Figure 3: Excerpt from ssh/keys.go

Nominally, p and q are prime, q divides $p - 1$, g and y are integers modulo p and have order q. The decoding function verifies none of these properties; it does not even check whether the values are positive. The verification function, in Go’s standard library (`crypto/dsa.Verify()`) is given the signature itself (two integers (r, s) of up to 160 bits each) and first verifies that $0 < r, s < q$ (hence q must be positive), the bit length of q is a multiple of 8, and s is invertible modulo q; however, nothing checks that q is not a very large integer. The verification routines entails computing two integer values u_1 and u_2 modulo q (both are typically as large as q itself), then two modular exponentiations that use $|p|$ as modulus, and u_1 and u_2 as exponents, respectively. The cost of these exponentiations is linear in the size of the exponents (and quadratic in the size of the modulus, which is here fixed at 1024 bits), so that a large q value implies a large CPU usage. When the size of q is slightly above 2

1. Cost growth is quadratic, but an extra threshold occurs at some modulus size between 32000 and 130000 bits; cost gets noticeably higher when the threshold is crossed. The threshold depends on the actual value of the signature, as an integer, and the cost is highest when the signature is a *small* integer, in particular zero; this threshold is *probably* a memory allocation effect, the short value making the implementation shrink and re-expand some intermediate values. For a very large modulus, the quadratic cost of Montgomery multiplication dominates.

million bits (the maximum size that can still fit in a packet sent over the wire), the verification will use up to **2.1 seconds** worth of CPU time on the same Intel x86 CPU as was used as example for the case of RSA.

These large CPU costs of signature verification can be leveraged for a denial-of-service attack subject to some conditions on the protocol deployment context:

- **Verification of user authentication by the server:** when the server uses key-based user authentication, the public key conveyed by the client is first sent to the application-provided callback `PublicKeyCallback()` (declared in *ssh/server.go*, line 131); the server will proceed with the signature verification only if that callback declared itself happy with the user's public key. However, a second callback `VerifiedPublicKeyCallback()` can be set (*ssh/server.go*, line 145), to be called *after* the signature verification. An application may elect to actually validate the user's public key only after that signature verification has been performed (if validating a user's public key is expensive, e.g. it entails some database lookups, then such delaying of the validation can make sense as part of mitigations of DoS attacks that target this validation). If the application is configured in such a way, then the DoS attacks based on invalid public keys, as described above, can be applied to the server.
- **Verification of the server (host) key by the client:** the client obtains the host public key and the host's signature value as part of the key exchange response (`SSH_MSG_KEXDH_REPLY`). The client performs the signature verification *before* invoking the application-provided callback which validates the host public key: in *ssh/handshake.go*, the `handshakeTransport.client()` function calls `verifyHostKeySignature()` first (line 837), and the `hostKeyCallback()` callback function only afterwards (line 841). Therefore, the DoS previously described can be leveraged by a malicious server. DoS attacks on clients usually have a lesser impact, because clients traditionally try only once or twice to connect before giving up, and cannot be made to attempt connections repeatedly and at high rate. However, the SSH implementation is a *library* and the SSH notions of "client" and "server" do not necessarily match the overall roles of the involved parties in the system that uses SSH as part of its communication protocols.
- **Verification of signatures on certificates:** the `CertChecker` type (*ssh/certs.go*, line 624) organizes certificate validation and contains application-provided callbacks to validate that a certificate issuer public key is acceptable. These callbacks are `IsHostAuthority()` and `IsUserAuthority()`, and are meant for the cases of certificates for hosts (verified by clients) and users (verified by servers), respectively; they are invoked by the validation functions (`CertChecker.CheckHostKey()` and `CertChecker.Authenticate()`) *before* proceeding to do the cryptographic signature verification itself with `CertChercker.CheckCert()`. While prior key validation may protect systems from the DoS described above, an application may decide to delay key validation for the same reasons as in verification of the user authentication (see previously), in which case the DoS conditions may apply again. Moreover, the `CertChecker.CheckCert()` function is part of the public API and can be invoked directly by the application with no prior key validation step.

In the first two contexts (verification of the host key or of the user key), an additional filter is the configuration of the allowed algorithms: RSA or DSA signature verification will be invoked only if the corresponding algorithm is part of the applied configuration. RSA is part of the default list of supported algorithms, but DSA is not; DSA-related DoS attacks on these operations may thus happen only if an application takes the unusual step of supporting DSA explicitly (probably for backward compatibility reasons). For signatures on certificates,

though, there is no such configuration: both RSA and DSA are always supported (the application-provided callbacks are supposed to filter against unwanted key types).

Recommendation

RSA keys should be systematically validated, upon decoding, to have values in reasonable ranges. i.e. that the public modulus n is positive, odd, and not larger than some maximum size (for instance 8192 bits). Similarly, DSA keys should be checked: all values should be positive; p must be odd and of an acceptable size (the current implementation already performs the size check); q must be odd, and lower than p (since it nominally divides $p - 1$); g and y must be greater than 1 but lower than p. All these checks are inexpensive and can thus be applied as part of the parsing functions.

Location

- *ssh/keys.go*, lines 463-483
- *ssh/keys.go*, lines 582-603

Retest Results

2026-04-10 – Fixed

The project team reviewed *0003-ssh-reject-RSA-keys-with-excessively-large-moduli.patch*, which updates `parseRSA()` to explicitly reject any modulus larger than 8192 bits. A regression test is included. Note that the recommendation to check that the modulus is both positive and odd is not implemented; however, rejecting oversized moduli is enough to fix the denial-of-service vulnerability.

0004-ssh-enforce-strict-limits-on-DSA-key-parameters.patch was also reviewed, which updates the `checkDSAParams()` function with the following checks:

- that q is exactly 160 bits (per FIPS 186-2),
- that g is less than p,
- that g is positive.

These checks also imply that p is positive, which in turn implies that $q < p$ (since p is verified to be a 1024-bit integer, and q is now verified to have size 160 bits).

Note that other recommended checks (e.g. that q is positive, or that $0 < y < p$) have not been implemented. Nevertheless, the checks that were implemented are sufficient to fully prevent the reported denial-of-service vulnerability. The regression test explicitly targets the case where q is larger than allowed.

In summary, this finding is considered *Fixed*, as the implemented length checks directly negate the CPU exhaustion attack.

2026-04-24 – Fixed

An updated patch *0004-ssh-enforce-strict-limits-on-DSA-key-parameters.patch* was reviewed which adds the recommended check that $0 < y < p$ in `parseDSA()`. Regression tests are included.

The prior fix remediated the denial-of-service concerns, and the revised patch more closely aligns with the complete recommendations in this finding.

2026-06-02 – Fixed
Rejection of oversized RSA keys was merged in https://go-review.googlesource.com/c/crypto/+/781641, additional DSA checks were merged in https://go-review.googlesource.com/c/crypto/+/781661, and the issue was publicly disclosed as CVE-2026-39829 and GO-2026-5018.

Medium

SSH Agent Client Drops Constraint Extensions

Overall Risk	Medium	**Finding ID**	NCC-E029268-ET6
Impact	Medium	**Category**	Cryptography
Exploitability	High	**Status**	Fixed

Impact

Applications that call `agent.NewClient(...).Add()` believing that custom constraint extensions (such as `restrict-destination-v00@openssh.com`) will be enforced actually load completely unrestricted keys into the remote agent. Any host that gains access to the forwarded agent can therefore use those private keys to authenticate to arbitrary services.

Description

When a key is loaded into a remote ssh-agent via the exported `ssh/agent` client, callers provide an `AddedKey` populated with `ConstraintExtensions` to express vendor-specific restrictions (e.g., OpenSSH's "restrict-destination" extension). The top-level API `client.Add` function implemented in *ssh/agent/client.go* and excerpted below, only serializes the lifetime and confirmation constraints (as highlighted in yellow) before marshalling the request.

```
833 // Add adds a private key to the agent. If a certificate is given,
834 // that certificate is added instead as public key.
835 func (c *client) Add(key AddedKey) error {
836   var constraints []byte
837
838   if secs := key.LifetimeSecs; secs != 0 {
839     constraints = append(constraints, ssh.Marshal(constrainLifetimeAgentMsg{secs})...)
840   }
841
842   if key.ConfirmBeforeUse {
843     constraints = append(constraints, agentConstrainConfirm)
844   }
845
846   cert := key.Certificate
847   if cert == nil {
848     return c.insertKey(key.PrivateKey, key.Comment, constraints)
849   }
850   return c.insertCert(key.PrivateKey, cert, key.Comment, constraints)
851 }
```

Figure 4: Excerpt of `Add()` from ssh/agent/client.go

On the receiving end, the agent's `parseConstraints` / `setConstraints` pipeline in *ssh/agent/server.go* is built to look for those extension identifiers so it can populate `AddedKey.ConstraintExtensions` when keys arrive:

```
246 func parseConstraints(constraints []byte) (lifetimeSecs uint32, confirmBeforeUse bool,
    ↳ extensions []ConstraintExtension, err error) {
247   for len(constraints) != 0 {
248     switch constraints[0] {
249     case agentConstrainLifetime:
250       if len(constraints) < 5 {
251         return 0, false, nil, io.ErrUnexpectedEOF
252       }
```

```
253         lifetimeSecs = binary.BigEndian.Uint32(constraints[1:5])
254         constraints = constraints[5:]
255       case agentConstrainConfirm:
256         confirmBeforeUse = true
257         constraints = constraints[1:]
258       case agentConstrainExtension, agentConstrainExtensionV00:
259         var msg constrainExtensionAgentMsg
260         if err = ssh.Unmarshal(constraints, &msg); err != nil {
261           return 0, false, nil, err
262         }
263         extensions = append(extensions, ConstraintExtension{
264           ExtensionName:    msg.ExtensionName,
265           ExtensionDetails: msg.ExtensionDetails,
266         })
267         constraints = msg.Rest
268       default:
269         return 0, false, nil, fmt.Errorf("unknown constraint type: %d", constraints[0])
270       }
271     }
272     return
273 }
```

Figure 5: Excerpt of `parseConstraints()` from ssh/agent/server.go

Because the client never emits those attributes, the agent will always see an empty `ConstraintExtensions` slice, regardless of what the caller requested.

A potential scenario is an administrator who forwards their agent into an untrusted jump host but tries to limit the forwarded key with `restrict-destination-v00@openssh.com`. Due to this bug, the restriction never reaches the agent; any user on that jump host (or any compromised process) can make the agent sign arbitrary challenges and pivot into additional production hosts.

Recommendation

When building the constraint payload in `client.Add`, iterate over `key.ConstraintExtensions` and, for each entry, append an `agentConstrainExtension` (and potentially the legacy `agentConstrainExtensionV00` for compatibility) byte followed by the marshalled `constrainExtensionAgentMsg`. Consider returning an error if unsupported extensions are requested to avoid silently downgrading security expectations. Consider adding a round-trip style regression test to ensure no attributes are lost.

Location

The `Add()` function implemented in *ssh/agent/client.go*

Retest Results

2026-04-10 – Fixed

The project team reviewed *0007-ssh-agent-preserve-constraint-extensions-when-adding.patch*, which updates the client `Add()` function to correctly append the `key.ConstraintExtensions`, as recommended, along with a regression test.

0008-ssh-agent-don-t-accept-keys-with-unsupported-constra.patch was also reviewed, which updates the keyring `Add()` function to reject any key with constraint extensions present. Tests were updated to validate this behavior, and the prior regression test was removed, as there are no supported extensions that can be tested.

In summary, this finding is considered *Fixed*.

2026-04-24 – Fixed
An updated *0007-ssh-agent-preserve-constraint-extensions-when-adding.patch* was reviewed that adds additional testing that better enforces the expected behavior.

2026-06-02 – Fixed
Reviewed patches were merged in https://go-review.googlesource.com/c/crypto/+/778640 and https://go-review.googlesource.com/c/crypto/+/778641, and publicly disclosed as CVE-2026-39832 and GO-2026-5006.

Medium

Nil Dereference Panic on `c.IsUserAuthority` and `c.IsHostAuthority`

Overall Risk	Medium	**Finding ID**	NCC-E029268-HHY
Impact	High	**Category**	Configuration
Exploitability	Medium	**Status**	Fixed

Impact

A server that wires the `CertChecker.Authenticate()` function into a `ServerConfig.PublicKeyCallback` but does not also set the `IsUserAuthority` field can be crashed by an unauthenticated client that sends a certificate. A similar issue applies to the `IsHostAuthority` field referenced by the `CertChecker.CheckHostKey()` function. The panics may drop existing sessions and prevent new connections.

Description

As noted in the comments preceding the `Authenticate()` function implemented in the *ssh/certs.go* source file and excerpted below, a server may use this function as a `ServerConfig.PublicKeyCallback` to take advantage of `CertChecker` features (critical option parsing, revocation hooks, or the `UserKeyFallback` hook for legacy keys). In this configuration, the attacker-controlled entry point becomes the public-key authentication request processed by the `serverAuthenticate()` function in the *ssh/server.go* source file, which passes the client-supplied key blob straight into the `config.PublicKeyCallback`.

When the client provides a certificate key, the `Authenticate()` follows the certificate path flow shown below.

```
364 // Authenticate checks a user certificate. Authenticate can be used as
365 // a value for ServerConfig.PublicKeyCallback.
366 func (c *CertChecker) Authenticate(conn ConnMetadata, pubKey PublicKey) (*Permissions,
    ↳ error) {
367   cert, ok := pubKey.(*Certificate)
368   if !ok {
369     if c.UserKeyFallback != nil {
370       return c.UserKeyFallback(conn, pubKey)
371     }
372     return nil, errors.New("ssh: normal key pairs not accepted")
373   }
374
375   if cert.CertType != UserCert {
376     return nil, fmt.Errorf("ssh: cert has type %d", cert.CertType)
377   }
378   if !c.IsUserAuthority(cert.SignatureKey) {
379     return nil, fmt.Errorf("ssh: certificate signed by unrecognized authority")
380   }
381
382   if err := c.CheckCert(conn.User(), cert); err != nil {
383     return nil, err
```

```
384     }
385
386     return &cert.Permissions, nil
387 }
```

Figure 6: Excerpt of `Authenticate()` from ssh/certs.go

The `IsUserAuthority` field of `CertChecker` defaults to `nil`, and there is no guard before the call. Note that the `UserKeyFallback()` function is only evaluated when the key is **not** a certificate, so servers that only intended to support non-certificate keys by providing a `UserKeyFallback()` (and therefore left `IsUserAuthority` unset) will instead dereference a `nil` function pointer when the attacker (unexpectedly) supplies any certificate.

An almost exact replica of this issue is also present in the adjacent `CheckHostKey()` function in the same source file. In this instance, the `c.IsHostAuthority` field may be invoked on a `nil` function reference.

Recommendation

Defensively handle the absence of authority callbacks before invoking them. In `CertChecker.Authenticate()`, check `c.IsUserAuthority` and immediately return error on `nil`. Similarly check `c.IsHostAuthority` in `CertChecker.CheckHostKey()` and immediately return an error on `nil`.

Location

`Authenticate()` and `CheckHostKey()` in *ssh/certs.go*

Retest Results

2026-04-17 – Fixed

The project team reviewed *0011-ssh-fix-panic-when-authority-callbacks-are-nil.patch*, which adds the recommended nil checks in `Authenticate()` and `CheckHostKey()`.

In summary, this finding is considered *Fixed*.

2026-06-02 – Fixed

The reviewed patch was merged in https://go-review.googlesource.com/c/crypto/+/781660 and publicly disclosed as CVE-2026-39835 and GO-2026-5015.

Medium

Hang on Unsolicited Global Request Replies

Overall Risk	Medium	**Finding ID**	NCC-E029268-WNA
Impact	Low	**Category**	Denial of Service
Exploitability	High	**Status**	Fixed

Impact

A malicious peer can freeze every channel multiplexed over their own connection by sending unsolicited `SSH_MSG_REQUEST_SUCCESS` or `SSH_MSG_REQUEST_FAILURE` messages. Once the shared global-response buffer fills, the connection's main `mux` goroutine stops reading network packets, so sessions, port forwards, and keepalive loops all hang until the transport is torn down. This only impacts the peer's connection, not others.

Description

The `NewServerConn()` function implemented in *ssh/server.go* starts a new SSH server given a `ServerConfig` and underlying transport. As expected, that function soon calls the `serverHandshake()` function which performs key exchange and user authentication before instantiating a `newMux()` as implemented in the *ssh/mux.go* source file. The `mux` struct contains a `globalResponses` field with a channel of depth 1.

Applications interact with **outbound** global requests through the exported `Conn.SendRequest()` API defined in the *ssh/connection.go* source file, which directly calls `mux.SendRequest()`. When the `wantReply` argument is true, the `SendRequest()` function pushes the request on the wire via `sendMessage()` and then waits for a reply pulled from `m.globalResponses`.

```
140 func (m *mux) SendRequest(name string, wantReply bool, payload []byte) (bool, []byte,
    ↳ error) {
141   if wantReply {
142     m.globalSentMu.Lock()
143     defer m.globalSentMu.Unlock()
144   }
145
146   if err := m.sendMessage(globalRequestMsg{
147     Type:      name,
148     WantReply: wantReply,
149     Data:      payload,
150   }); err != nil {
151     return false, nil, err
152   }
153
154   if !wantReply {
155     return false, nil, nil
156   }
157
158   msg, ok := <-m.globalResponses
159   if !ok {
160     return false, nil, io.EOF
161   }
162   switch msg := msg.(type) {
163   case *globalRequestFailureMsg:
```

```
164     return false, msg.Data, nil
165   case *globalRequestSuccessMsg:
166     return true, msg.Data, nil
167 ...
```

Figure 7: Excerpt of `SendRequest()` from ssh/mux.go

The above design assumes a `wantReply`-related reader is aways awaiting the response. Now, consider **unsolicited incoming** replies from a malicious peer. Incoming packets are received inside the single `mux.loop()` goroutine that calls `onePacket()` that in turn calls `readPacket()` and passes global packets to `handleGlobalPacket()`. This latter function unconditionally pushes every `SSH_MSG_REQUEST_SUCCESS` or `SSH_MSG_REQUEST_FAILURE` onto the single-slot `globalResponses` buffer, even when the local side never asked for (or expected) a reply.

```
257 func (m *mux) handleGlobalPacket(packet []byte) error {
258   msg, err := decode(packet)
259   if err != nil {
260     return err
261   }
262
263   switch msg := msg.(type) {
264   case *globalRequestMsg:
265     m.incomingRequests <- &Request{
266       Type:      msg.Type,
267       WantReply: msg.WantReply,
268       Payload:   msg.Data,
269       mux:       m,
270     }
271   case *globalRequestSuccessMsg, *globalRequestFailureMsg:
272     m.globalResponses <- msg
273   default:
274     panic(fmt.Sprintf("not a global message %#v", msg))
275 ...
```

Figure 8: Excerpt of `handleGlobalPacket()` from ssh/mux.go

As a result, after the handshake, an adversarial peer can simply transmit two forged reply packets without any preceding request. The first fills the length 1 `globalResponses` channel which is never drained because no goroutine is reading from it. The very next reply packet blocks in `handleGlobalPacket()` while attempting to send on the full channel. Because `handleGlobalPacket()` executes inside `mux.loop()`, the entire connection state machine stops servicing every other SSH message, deadlocking all active channels and keepalives. No protocol error or timeout clears the condition, so the DoS persists until the application notices the hang and tears down the transport.

Recommendation

Treat unmatched `SSH_MSG_REQUEST_SUCCESS` and `SSH_MSG_REQUEST_FAILURE` frames as protocol errors instead of enqueueing them. For example, guard the send with a non-blocking `select` that drops or logs extra replies when no request is outstanding, or add tracking so that unsolicited responses trigger immediate connection teardown. Additionally, consider draining stale entries before issuing a new global request so that responses are always correlated with the request that triggered them.

Location

`handleGlobalPacket()` in *ssh/mux.go*

Retest Results

2026-04-10 – Fixed

The project team reviewed *0005-ssh-fix-deadlock-on-unexpected-global-responses.patch*, which updates the `handleGlobalPacket()` function to return an error on unmatched `SSH_MSG_REQUEST_SUCCESS` and `SSH_MSG_REQUEST_FAILURE` frames, as recommended, along with a regression test.

Furthermore, the same patch implements the recommendation to drain stale entries in `SendRequest()`, along with a regression test.

In summary, this finding is considered *Fixed*.

2026-04-24 – Fixed

An updated *0005-ssh-fix-deadlock-on-unexpected-global-responses.patch* was reviewed:

> The previous version only made the send to `globalResponses` non-blocking, which stopped the deadlock but still allowed a peer to stage a spurious response in the buffer so the next legitimate `SendRequest` consumed the wrong reply. The revised patch adds a `globalSentPending atomic.Bool` gate: `handleGlobalPacket()` now drops responses when no `SendRequest` is in flight.

These changes further harden the message handling above what was recommended in this finding.

2026-04-24 – Fixed

In addition to the fixes documented above, *0013-ssh-fix-deadlock-on-unexpected-channel-responses.patch* was provided as a fix to related issue; see client response below.

2026-06-02 – Fixed

The reviewed patches were merged as https://go-review.googlesource.com/c/crypto/+/781640 and https://go-review.googlesource.com/c/crypto/+/781664, along with public disclosure as CVE-2026-39830 and GO-2026-5017.

Client Response

As part of remediation, Geomys provided additional context on a similar issue that was discovered and patched.

`channel.handlePacket` was dispatching `channelRequestSuccess` / `Failure` to `ch.msg` unconditionally via the default arm of the type switch. Because `ch.msg` is bounded (`chanSize`), a peer sending a burst of unsolicited channel-request responses on an open idle channel fills the buffer and blocks the mux read loop on the next send, which in turn backs up `readLoop` on `t.incoming`; closing the underlying `net.Conn` then does not unblock anything, so `Close()` returns promptly while `Wait()` hangs and the mux/`readLoop` / `kexLoop` goroutines leak permanently. The fix mirrors 0005: a per-channel `sentRequestPending atomic.Bool` gate, non-blocking send when open, silent drop when closed, drain-before-send in the `SendRequest` path. This matches OpenSSH, which silently ignores channel confirm messages that do not match a pending request.

Medium

Maliciously Crafted Private Keys Can Trigger Panics or Very High CPU Usage

Overall Risk	Medium	**Finding ID**	NCC-E029268-GDJ
Impact	High	**Category**	Data Validation
Exploitability	Low	**Status**	Fixed

Impact

A maliciously crafted private key, when loaded or used by the SSH implementation, may trigger a panic, or induce high CPU usage (several months worth of CPU time). This may constitute a denial-of-service in some usage contexts (e.g. cloud-based applications in a bring-your-own-key model).

Description

Private keys for signing purposes can be loaded into the SSH library, on both the client and the server, by decoding the keys from PEM-encoded files. Maliciously crafted private key files can induce unwanted behaviour such as panics or very high CPU usage, in at least the following ways. All functions referenced below are from source file *ssh/keys.go*.

Panic with a DSA key using a non-standard subgroup size: A DSA private key works in a multiplicative subgroup specified by three big integers p, q and g: p is a big prime integer, q is a smaller prime that divides $p - 1$, and g is a generator of the multiplicative subgroup of order q modulo p. The `ParseDSAPrivateKey()` function (line 1354) decodes these values as generic big integers from an ASN.1 format, and stores them in a `crypto/dsa.PrivateKey` structure without any other checks.

Before being used, the key object is converted to a `Signer` instance by the `newDSAPrivateKey()` function (line 1109), which first calls `checkDSAParams()` (line 569). This last function *only* verifies that the modulus p has a bit length of 1024 bits. As per the FIPS 186-2 standard, p must have length exactly 1024 bits, and q must have length exactly 160 bits; the latter property is *not* checked, so that a non-compliant (maliciously crafted) DSA private key with a subgroup order q larger than 160 bits will be successfully decoded.

A signature is obtained with either the `dsaPrivateKey.SignWithAlgorithm()` function (line 671) or the `wrappedSigner.SignWithAlgorithm()` function (line 1146). Similar code is used in both, and have the same issue; we show here an excerpt from the first function:

```
683     r, s, err := dsa.Sign(rand, k.PrivateKey, digest)
684     if err != nil {
685         return nil, err
686     }
687
688     sig := make([]byte, 40)
689     rb := r.Bytes()
690     sb := s.Bytes()
691
692     copy(sig[20-len(rb):20], rb)
693     copy(sig[40-len(sb):], sb)
```

Figure 9: Excerpt from ssh/keys.go

The signature is obtained as two big integers r and s, which are about as large as the subgroup order q. The two integers are converted to sequences of bytes (with unsigned big-endian convention) on lines 689-690, then copied into the 40-byte `sig` buffer on lines 692-693. If q is larger than 160 bits, then, with high probability, `rb` will have length 21 or more bytes, which will lead to a negative slice start index (`20-len(rb)`), which will trigger a runtime panic.

This issue can occur only if DSA signature generation is part of the algorithms supported by the SSH configuration; the default configuration does not include DSA support, which limits the scope of this issue.

Oversized bcrypt round number: when processing a passphrase-protected private key in OpenSSH format, the password-based key derivation function is `bcrypt_pbkdf`, as used in the OpenBSD operating system. This function uses a salt and a configurable number of rounds, meant to enforce slow processing to discourage brute force attacks on the passphrase. These two parameters are decoded from the file itself:

```
1403        var opts struct {
1404           Salt   string
1405           Rounds uint32
1406        }
1407        if err := Unmarshal([]byte(kdfOpts), &opts); err != nil {
1408           return nil, err
1409        }
1410
1411        k, err := bcrypt_pbkdf.Key(passphrase, []byte(opts.Salt), int(opts.Rounds), 32+16)
```

Figure 10: Excerpt from ssh/keys.go

As shown above, the number of rounds is decoded as a `uint32`; a normal value is 16 (this is what the SSH implementation uses when encoding a key with passphrase protection), but there is no attempt at validating the received value, which can thus be as high as 4294967295. On a 64-bit system, such a value will be preserved by the conversion to `int`, and passed to the internal `bcrypt_pbkdf` implementation. Since the key derivation time is proportional to the number of rounds, such a high value will occupy the CPU for a long time (on an Intel i5-8259U at 2.3 GHz, no frequency scaling, running 64-bit Linux and Go 1.25.5, total estimated time is **588 days**), which could be leveraged for a denial-of-service attack if the attacker can upload their own private key file in a remote service that will use that key in SSH. Note that the issue does not depend on the type or size of the private key which is purportedly protected.

Oversized RSA modulus: when decoding an RSA Key pair in the OpenSSH-specific format, the `parseOpenSSHPrivateKey()` function decodes the source bytes with the `openSSHRSAPrivateKey` type:

```
1571        var key openSSHRSAPrivateKey
1572        if err := Unmarshal(pk1.Rest, &key); err != nil {
1573           return nil, err
1574        }
1575
1576        if err := checkOpenSSHKeyPadding(key.Pad); err != nil {
1577           return nil, err
1578        }
1579
```

```
1580        pk := &rsa.PrivateKey{
1581            PublicKey: rsa.PublicKey{
1582                N: key.N,
1583                E: int(key.E.Int64()),
1584            },
1585            D:      key.D,
1586            Primes: []*big.Int{key.P, key.Q},
1587        }
1588
1589        if err := pk.Validate(); err != nil {
```

Figure 11: Excerpt from ssh/keys.go

A private RSA key with two prime factors and leveraging the "CRT" format nominally includes not only the prime factors p and q, but also the CRT coefficient $1/q \bmod p$ which allows computing private key operations modulo the half-size p and q rather than the full-size modulus n. This CRT coefficient is part of the encoding format, and is unmarshalled into the `Iqmp` field of the `openSSHRSAPrivateKey` structure; however, the implementation then discards the received value, and builds the RSA private key in-memory object with only p and q (as well as the modulus n, public exponent e and private exponent d). The call to `Validate()` on line 1589 will indirectly lead to the internal Go backend for RSA private operations, which will recompute the CRT coefficient. This modular inversion is performed with Fermat's "little theorem", i.e. raising q to the power $p - 2$ modulo p. If p is a large integer then that operation can be very expensive; its cost is *cubic* in the size of p.

There is no check on the sizes of the integers prior to that call. The RSA backend will verify that p and q are odd, and that their product is n; thus, a maliciously crafted private key file trying to make p a very large integer will also need to make n as large as p (q can be kept small). If the size of p is 65536 bits (which would fit in a file of about 22 kilobytes, accounting for large p and n, and the Base64 encoding), then the computation of the CRT coefficient requires about **7.3 minutes** on a test 2.3 GHz Intel CPU; increasing the sizes of p and n to 1048576 bits (file size of about 350 kilobytes) would raise that time to close to **21 days** (possibly more, because of cache effects). As in the case of the bcrypt issue previously described, this could be used for a denial-of-service attack. Also note that this happens upon *decoding* the private key file, independently of the SSH implementation configuration: the issue is still present even if the implementation is configured not to ever use RSA signatures.

Recommendation

The following actions are recommended:

- Enforce standard-compliant sizes for all DSA parameters in `checkDSAParams()`, including the subgroup order q, which should be an integer of exactly 160 bits in length (such an enforcement would also help with the DSA variant of the issue described in finding "Abnormal RSA and DSA Public Keys Incur High Cost for Signature Verification").
- Apply a strict maximum limit on the number of rounds for bcrypt-based key derivation. The default value is 16, both in the Go implementation and in OpenSSH; some users may have elected to use higher values for a variety of reasons, so that maybe the Go implementation should allow up to, e.g., 256 rounds.

- Apply a strict maximum size on RSA keys, e.g. 8192 bits for the modulus n and 4096 bits for each prime factor.

Location

ssh/keys.go

Retest Results

2026-04-17 – Fixed

The project team reviewed the change request at: https://go-review.googlesource.com/c/crypto/+/782422/2. This CR adds an upper limit to the number of supported bcrypt rounds, as recommended.

https://go-review.googlesource.com/c/crypto/+/782420/2 was also reviewed, which enforces an upper bound on the size of the RSA primes and modulus.

0004-ssh-enforce-strict-limits-on-DSA-key-parameters.patch was also reviewed, which updates the `checkDSAParams()` function. See retest notes in finding "Abnormal RSA and DSA Public Keys Incur High Cost for Signature Verification" for additional details.

In summary, this finding is considered *Fixed*.

Medium

Insufficient Validation of `ed25519` Private Key

Overall Risk	Medium	**Finding ID**	NCC-E029268-BWW
Impact	High	**Category**	Data Validation
Exploitability	Medium	**Status**	Fixed

Impact

An application that accepts ed25519 private keys from an untrusted source and provides them to the `client.Add()` function can be reliably crashed, as supplying fewer than 32-bytes causes the called `insertKey()` and/or `insertCert()` functions to panic and may terminate the process.

Description

When providing an ed25519 key to the `client.Add()` function, no length validation happens before execution is transferred to either the `client.insertKey()` or `client.insertCert()` functions.

Inside the `insertKey()` function the ed25519 branch(es) assume the byte slice is made of a 32-byte seed followed by a 32-byte public key and slices it at an absolute offset. Because the `ed25519.PrivateKey` is not a fixed-size type but instead a `[]byte`, passing fewer than 32 bytes makes the expression `[]byte(k)[32:]` (or its pointer variant) panic with `slice bounds out of range`. The slicing is excerpted below.

```
// Insert adds a private key to the agent.
func (c *client) insertKey(s interface{}, comment string, constraints []byte) error {
  var req []byte
  switch k := s.(type) {
  case *rsa.PrivateKey:
  ...
  case *dsa.PrivateKey:
  ...
  case *ecdsa.PrivateKey:
  ...
  case ed25519.PrivateKey:
    req = ssh.Marshal(ed25519KeyMsg{
      Type:        ssh.KeyAlgoED25519,
      Pub:         []byte(k)[32:],
      Priv:        []byte(k),
      Comments:    comment,
      Constraints: constraints,
    })
  // This function originally supported only *ed25519.PrivateKey, however the
  // general idiom is to pass ed25519.PrivateKey by value, not by pointer.
  // We still support the pointer variant for backwards compatibility.
  case *ed25519.PrivateKey:
    req = ssh.Marshal(ed25519KeyMsg{
      Type:        ssh.KeyAlgoED25519,
      Pub:         []byte(*k)[32:],
      Priv:        []byte(*k),
      Comments:    comment,
      Constraints: constraints,
    })
```

```
  default:
    return fmt.Errorf("agent: unsupported key type %T", s)
  }
...
}
```

Figure 12: Excerpt of `insertKey()` from ssh/agent/client.go

The exact highlighted statements are also present in the subsequent `insertCert()` function that is reachable along the same path.

No recovery is performed, so unless every caller wraps `client.Add()` in a `recover`, any malformed ed25519 key shorter than 32 bytes reliably terminates the process.

Recommendation

Before slicing the ed25519 key material, in all instances explicitly validate that the underlying byte slice length is exactly `ed25519.PrivateKeySize` (64 bytes) and return an error when it is not.

Location

The `insertKey()` and `insertCert()` functions in *ssh/agent/client.go*

Retest Results

2026-04-17 – Fixed

The project team reviewed *0009-ssh-agent-fix-panic-on-malformed-Ed25519-keys-in-cli.patch*, which adds the recommended length checks in the affected functions, for both pointer and slice variants.

In summary, this finding is considered *Fixed*.

2026-06-02 – Fixed

The reviewed patch was merged in https://go-review.googlesource.com/c/crypto/+/782423 and publicly disclosed as CVE-2026-46598 and GO-2026-5033.

Medium

Insufficient Validation of Userauth Messages Before Signing

Overall Risk	Medium	**Finding ID**	NCC-E029268-MN2
Impact	Undetermined	**Category**	Data Validation
Exploitability	Medium	**Status**	Fixed

Impact

An agent may sign a message that is not actually a user authentication request for the claimed user public key. This could allow, for example, a compromised machine to which an agent has been forwarded to obtain unauthorized signatures on specially crafted messages.

Description

When agent forwarding is used, the agent receives user authentication requests on a designated channel, which is handled in `ServeAgentV2()` (*ssh/agent/server.go*). This function subsequently calls `(*server).processRequestBytes()` and `s.processRequest()` on the received bytes. The first byte indicates the command – in this case, `agentSignRequest` (`SSH_AGENTC_SIGN_REQUEST`). The bytes are unmarshaled into a `signRequestAgentMsg`:

```
118 type signRequestAgentMsg struct {
119   KeyBlob []byte `sshtype:"13"`
120   Data    []byte
121   Flags   uint32
122 }
```

Figure 13: ssh/agent/client.go

Then, the agent's `Sign()` function is called on the request's `KeyBlob` (as the desired signing key), and the request's `Data` (as the message to sign).

In `(*keyring).Sign()`, if a match `k` for the given `KeyBlob`'s public part is found in the keyring, then `k.checkForSigning()` is called on the `Data`. This function parses the `Data` using the helper function `ParsePublicKeyUserAuthRequest()`.

`ParsePublicKeyUserAuthRequest()` first parses the `Data` into a session ID and `userAuthRequestMsg`:

```
88 type userAuthRequestMsg struct {
89   User    string `sshtype:"50"`
90   Service string  // "ssh-connection"
91   Method  string
92   Payload []byte `ssh:"rest"`
93 }
```

Figure 14: ssh/messages.go

Then, it parses the `Payload` to extract an algorithm identifier and a public key. With all of this parsed data, it forms a `PublicKeyUserAuthRequest`:

```
12 // PublicKeyUserAuthRequest represents the parsed fields of an
13 // SSH_MSG_USERAUTH_REQUEST packet.  See RFC 4462 section 3.2.
14 type PublicKeyUserAuthRequest struct {
15   User      string
16   Service   string
17   Method    string
```

```
18    SessionID []byte
19    Algorithm string
20    PublicKey PublicKey
21    HostKey   PublicKey
22  }
```

Figure 15: ssh/messages.go

Back in `checkForSigning()`, `k.isPermitted()` is called on the `User` in the parsed `PublicKeyUserAuthRequest`; it verifies that any session binding or destination constraints are obeyed. Lastly, in `checkForSigning()`, there are checks that the included `SessionID` and `HostKey` match the ones expected by the agent.

Back in `(*keyring).Sign()`, either `Sign()` or `SignWithAlgorithm()` is called on the `Data`; no more checks are performed on the message to be signed.

Several important checks on the data-to-be-signed are missing, so the agent ends up signing a message that it has not fully verified. This means that it may be possible for a user with legitimate access to craft a special message that seems to be a user authentication request but is actually something else (not authorized by the agent) that bypasses the key's constraints.

The following checks are not present in the *agent* package code:

- The `PublicKey` in the `PublicKeyUserAuthRequest` should match (both in value and type) the expected key from the outer `KeyBlob`.
- The `Algorithm` in the `PublicKeyUserAuthRequest` should match the type of the expected key (from the outer `KeyBlob`).

In OpenSSH, these checks are present in `parse_userauth_request()` (*ssh/ssh-agent.c*):

```
if (t != SSH2_MSG_USERAUTH_REQUEST ||
    sig_follows != 1 ||
    strcmp(service, "ssh-connection") != 0 ||
    !sshkey_equal(expected_key, mkey) ||
    sshkey_type_from_name(pkalg) != expected_key->type) {
  r = SSH_ERR_INVALID_FORMAT;
  goto out;
}
```

Figure 16: Excerpt from `parse_userauth_request()` in ssh/ssh-agent.c

In summary, the agent may sign a message with two unverified fields, highlighted in the following excerpt from RFC 4252.

```
The value of 'signature' is a signature by the corresponding private
key over the following data, in the following order:

   string    session identifier
   byte      SSH_MSG_USERAUTH_REQUEST
   string    user name
   string    service name
```

```
      string    "publickey"
      boolean   TRUE
      string    public key algorithm name
      string    public key to be used for authentication
```

Figure 17: Excerpt from Section 7 Public Key Authentication Method: "publickey" of RFC 4252

The impact of this finding depends on what else the key is used for. If used only for SSH user authentication, the impact is limited since signature verification should check that the public key contained in the message is the same key that was used for signature verification. However, humans frequently re-use keys for multiple purposes, and it is possible that the lack of strict message validation before signing can be leveraged to gain unauthorized access in another protocol/system.

Recommendation

Add the extra checks to fully validate user authentication request messages before signing them.

Location

- Function `(*privKey).checkForSigning()` in *ssh/agent/keyring.go*
- Function `ParsePublicKeyUserAuthRequest()` in *ssh/messages.go*

Retest Results

2026-04-17 – Fixed

The project team reviwed commit `48280e0` which adds the recommended checks in `checkForSigning()`. Regression tests are included. It is also noted that this finding affected code that was not yet part of the upstream repository.

In summary, this finding is considered *Fixed*.

Low

Large Channel Writes May Stall Forever

Overall Risk	Low	**Finding ID**	NCC-E029268-NTF
Impact	Low	**Category**	Denial of Service
Exploitability	Low	**Status**	Fixed

Impact

If a client or server tries to send 4 GB of data or more to a channel in a single function call, then only an infinite stream of empty packets is sent. This might constitute a denial-of-service attack in contexts where an attacker can trigger such a call in a SSH connection between two target systems.

Description

The *crypto/ssh/channel.go* file implements the channel `WriteExtended()` function, which is used for the `Write()` calls on both extended and non-extended channels. This function splits the request (a slice of bytes called `data`) into individual messages that comply with the negotiated maximum payload lengths and current window size in the flow control protocol:

```
253     for len(data) > 0 {
254         space := min(ch.maxRemotePayload, len(data))
255         if space, err = ch.remoteWin.reserve(space); err != nil {
256             return n, err
257         }
258         if want := headerLength + space; uint32(cap(packet)) < want {
259             packet = make([]byte, want)
260         } else {
261             packet = packet[:want]
262         }
263
264         todo := data[:space]
265
266         packet[0] = opCode
267         binary.BigEndian.PutUint32(packet[1:], ch.remoteId)
268         if extendedCode > 0 {
269             binary.BigEndian.PutUint32(packet[5:], uint32(extendedCode))
270         }
271         binary.BigEndian.PutUint32(packet[headerLength-4:], uint32(len(todo)))
272         copy(packet[headerLength:], todo)
273         if err = ch.writePacket(packet); err != nil {
274             return n, err
275         }
276
277         n += len(todo)
278         data = data[len(todo):]
279     }
```

Figure 18: Excerpt from ssh/channel.go

The `min()` function is used to compute the size of the data to send at each iteration. That function is implemented in the same file:

```
134 func min(a uint32, b int) uint32 {
135     if a < uint32(b) {
136         return a
137     }
138     return uint32(b)
139 }
```

Figure 19: Excerpt from ssh/channel.go

The comparison is done between the maximum payload size, which has type `uint32`, and the number of bytes remaining in the source slice, which has type `int` (as returned by the `len()` builtin function). On a 64-bit architecture, `int` a 64-bit type, and its values can exceed the range representable in a `uint32`. The `min()` function above systematically converts the `int` length to a `uint32` value, which may truncate it. In particular, if the `Write()` function on a channel is called with a `data` slice of size exactly 2^{32} bytes, then the second parameter to the `min()` call will be converted to a `uint32` of value zero, and `min()` will return zero. The rest of the loop in `WriteExtended()` will dutifully assemble and send a packet with its own header and padding and MAC, but with an empty payload. Since no data byte is actually sent, the length of `data` is unchanged, and the loop repeats indefinitely. In effect, network bandwidth is consumed, and both client and server spend CPU time to process the encryption and MAC on all packets, but no progress is done in sending the data.

More generally, if the source slice has size $2^{32}m + n$ bytes, for integers m and n such that $0 \leq n < 2^{32}$, then only the first n bytes of data are sent, and then the process hits the endless stalling condition described above.

Though modern systems are perfectly able to allocate arrays of 4 gigabytes or more, it is still an unusual occurrence, making this issue rarely exploitable in practice. One may envision an application that acts as a Web portal to receive files uploaded by users, then transfers such files to another server using the SSH protocol, with a single `Write()` call for the entire file; in that case, the attacker may upload a large file, which will make the portal and its backend server reach this stalling condition.

Recommendation

The `min()` function should compare its parameters (a `uint32` value, and an `int` value) by first converting both to `int64`, which would avoid the issue described here.

Location

crypto/ssh/channel.go, line 135

Retest Results

2026-04-10 – Fixed

The project team reviewed *0010-ssh-fix-infinite-loop-on-large-channel-writes-due-to.patch*, which adds and utilizes the `minPayloadSize()` function to correctly compare the payload size to the `maxRemotePayload` value, as recommended. Regression tests are included.

In summary, this finding is considered *Fixed*.

2026-04-24 – Fixed
An updated *0010-ssh-fix-infinite-loop-on-large-channel-writes-due-to.patch* was reviewed, which includes additional end-to-end testing and expanded function documentation for `minPayloadSize()`.

2026-06-02 – Fixed
The reviewed patch was merged in https://go-review.googlesource.com/c/crypto/+/781663 and publicly disclosed as CVE-2026-39834 and GO-2026-5020.

Low

SK* Signatures Ignore User Presence Flag

Overall Risk	Low	**Finding ID**	NCC-E029268-C6P
Impact	Low	**Category**	Cryptography
Exploitability	Medium	**Status**	Fixed

Impact

Applications may accept signatures involving `sk*` public keys even when the authenticator explicitly reports that no human was present. An attacker who can drive a hardware token remotely can therefore log in, without the touch confirmation that administrators deploying `sk*` keys intend to require. This effectively reduces the factor to a standard unattended private key.

Description

When a client authenticates with the public-key method, the `ServerConfig.PublicKeyCallback()` function ultimately calls the `PublicKey.Verify()` (interface) function on the received key. If the key type is `sk-ecdsa-sha2-nistp256@openssh.com` or `sk-ssh-ed25519@openssh.com`, the `parseSKECDSA()` or `parseSKEd25519()` functions return a `skECDSAPublicKey` or `skEd25519PublicKey` instance respectively, whose `Verify()` method is responsible for enforcing the U2F/FIDO semantics described in `ftp:` _openssh-10.2.tar.gz/PROTOCOL.u2f.

In the `skECDSAPublicKey.Verify()` function implemented in *ssh/keys.go* and excerpted below, the code parses the additional `skFields` payload but never checks the flags to ensure that the authenticator asserted user presence.

```
func (k *skECDSAPublicKey) Verify(data []byte, sig *Signature) error {
  ...
  var skf skFields
  if err := Unmarshal(sig.Rest, &skf); err != nil {
    return err
  }

  blob := struct {
    ApplicationDigest []byte `ssh:"rest"`
    Flags             byte
    Counter           uint32
    MessageDigest     []byte `ssh:"rest"`
  }{
    appDigest,
    skf.Flags,
    skf.Counter,
    dataDigest,
  }

  original := Marshal(blob)

  h.Reset()
  h.Write(original)
  digest := h.Sum(nil)

  if ecdsa.Verify((*ecdsa.PublicKey)(&k.PublicKey), digest, ecSig.R, ecSig.S) {
```

```
    return nil
  }
  return errors.New("ssh: signature did not verify")
}
```

Figure 20: Excerpt of `skECDSAPublicKey.Verify()` from ssh/keys.go

The `skEd25519PublicKey.Verify()` function implemented in the same source file repeats this exact logic with Ed25519 signatures.

In both cases `skf.Flags` is only copied into the hashed blob; no validation is performed, so the verifier does not enforce the `SSH_SK_USER_PRESENCE_REQD` bit is set. The bit exists so that servers can reject signatures produced without a touch.

As a result, any authenticator or middleware that is willing (or tricked) into signing without verifying presence can set the flag to zero and still be accepted by the application utilizing this code. This behavior downgrades `sk*` keys from "must touch" credentials to unattended keys, defeating the documented security property.

Recommendation

Add explicit validation of the user-presence bit in both `skECDSAPublicKey.Verify()` and `skEd25519PublicKey.Verify()`. Consider mirroring OpenSSH's verifier by defining a constant for `SSH_SK_USER_PRESENCE_REQD` (value `0x01` per *sk-api.h:25*) and return an error when `(skf.Flags & SSH_SK_USER_PRESENCE_REQD) == 0` before running the signature check.

Location

`skECDSAPublicKey.Verify()` and `skEd25519PublicKey.Verify()` in *ssh/keys.go*.

Retest Results

2026-04-10 – Fixed

The project team reviewed *0006-ssh-enforce-user-presence-verification-for-security-.patch*, which updates the `serverAuthenticate()` function to correctly handle the `"no-touch-required"` extension, along with the `skECDSAPublicKey` and `skEd25519PublicKey Verify()` functions to return an error when the user presence flags are not consistent when verifying a signature. Regression tests are included.

In summary, this finding is considered *Fixed*.

2026-04-24 – Fixed

An updated *0006-ssh-enforce-user-presence-verification-for-security-.patch* was reviewed that refactors the previous fix. This updated approach includes the following:

- Updated certificate validation to ignore the user presence flag in the certificate, thus matching OpenSSH behavior.
- `skECDSAPublicKey` and skEd25519PublicKey `updated to contain a` noTouchRequired bool`.
- Updated the `Verify()` functions to use this new `bool`.
- Several new regression tests.

The patch notes also emphasize the following:

> This change breaks backward compatibility for clients or keys that generate user-authentication signatures without the User Presence flag set. Previously those signatures were accepted by the server. They will now be rejected with "ssh:

> signature missing required user presence flag" unless the "no-touch-required" extension is explicitly granted to the session by the server callbacks, or carried by the user certificate.

2026-06-02 – Fixed
Confirmed as merged in https://go-review.googlesource.com/c/crypto/+/781662 and disclosed as CVE-2026-39831 and GO-2026-5019.

Low

Incorrect Handling of Usernames in Multi-Hop Constraint Validation with Binding

Overall Risk	Low	**Finding ID**	NCC-E029268-JGV
Impact	Low	**Category**	Cryptography
Exploitability	None	**Status**	Fixed

Impact

Forwarded authentication requests that should be allowed according to destination constraints are rejected.

Description

In *crypto/ssh*, an agent that receives a message of type `agentSignRequest` on the "auth-agent@openssh.com" channel, which is set up when agent forwarding is enabled, will ultimately call `(*keyring).Sign()`. This `Sign()` function performs several steps. First, it parses the `signRequestAgentMsg`'s `Data` into a `PublicKeyUserAuthRequest`:

```
895 // PublicKeyUserAuthRequest represents the parsed fields of an
896 // SSH_MSG_USERAUTH_REQUEST packet.  See RFC 4462 section 3.2.
897 type PublicKeyUserAuthRequest struct {
898     User      string
899     Service   string
900     Method    string
901     SessionID []byte
902     Algorithm string
903     PublicKey PublicKey
904     HostKey   PublicKey
905 }
```

Figure 21: Excerpt from ssh/messages.go

Next, if the specified signing key `privKey` has destination constraints, `(*privKey).isPermitted()` performs some checks on the username in the `PublicKeyUserAuthRequest` and the current `session`. The agent's `session` has the following format:

```
173 // Session represents the state of the current connection to the agent,
174 // specifically tracking the chain of session binds (hops) established via the
175 // "session-bind@openssh.com" extension.
176 type Session struct {
177   // Binds contains the list of hops recorded for this connection.
178   Binds []SessionBind
179   // BindsAttempted indicates if the client has attempted to send session-bind
180   // messages, even if the list is empty.
181   BindsAttempted bool
182 }
```

Figure 22: Excerpt from ssh/agent/client.go

The checks it performs on the `session` and `username` are:

1. If session binding has previously been attempted, then at least one bind has occurred (i.e., one hop has been recorded for this connection), where a `SessionBind` contains the following information:

```
94  // SessionBind represents a single hop in the agent forwarding chain, as defined
95  // by the "session-bind@openssh.com" extension in [PROTOCOL.agent].
96  type SessionBind struct {
97    HostKey    ssh.PublicKey
98    SessionID  []byte
99    Forwarding bool
100 }
```

Figure 23: Excerpt from ssh/messages.go

2. If there is at least one recorded session bind, every bind except the last one should have `Forwarding == true`.
3. For each bind, the `username` supplied to `isPermitted()`, the bind's `HostKey`, and the `HostKey` of the previous bind (if there was one) should be permitted according to `(*privKey).permittedByDestConstraints()`. Destinations constraints have the following format:

```
// DestinationConstraint defines a single rule allowing a key to be used
// from a specific source host to a specific destination host.
type DestinationConstraint struct {
   From HostIdentity
   To   HostIdentity
} // SNIP...

// HostIdentity represents a host and user specification within a constraint,
// including the expected host keys.
type HostIdentity struct {
   Username string
   Hostname string
   HostKeys []KeySpec
} // SNIP...

// KeySpec represents a specific key (and whether it is a CA) allowed for a host
// in a destination constraint.
type KeySpec struct {
   Key ssh.PublicKey
   CA  bool
}
```

Figure 24: Excerpts from ssh/agent/messages.go

This function iterates over all of the key's destination constraints and returns true if at least *one* constraint satisfies *all* of the following properties:

- If there is no `HostKey` specified for the previous bind, then the `From` part of the constraint should either have an empty username or an empty `HostKeys` list.
- Otherwise, the `HostKey` of the previous bind must match one of the `HostKeys` in the `From` constraint and the `HostKey` of the current bind must similarly match one of the `To` constraint's `HostKeys`.

- Either the `username` supplied to `isPermitted()` is `nil`, or the constraint's `To` part has an empty username, or `matchPattern(*username, constraint.To.Username)` returns true. This latter function performs recursive pattern matching with a maximum recursion depth of 64.

The last property is problematic: it always compares the username in the `PublicKeyUserAuthRequest` to the username in the `To` portion of a constraint, even when looking for a constraint that satisfies an intermediate hop in the connection. This check appears to indicate an implicit assumption that the usernames in a chain of destination restrictions are all identical, which is not necessarily the case. Instead, the `username` argument passed to `permittedByDestConstraints()` should be the username *for that hop*, or one should not be passed at all except for the final hop.

Comparison with OpenSSH's Implementation

In OpenSSH, the analogous functions are

- `identity_permitted()` (*usr.bin/ssh/ssh-agent.c*, line 461), which loops over the hops in the connection, and
- `permitted_by_dest_constraints()`, (*usr.bin/ssh/ssh-agent.c*, line 405), which is called for each hop, and tries to match the user **if one is specified**.

However, OpenSSH's `identity_permitted()` function passes the user parameter to `permitted_by_dest_constraints()` only for the last hop (see line 520). In *ssh/agent*, when `checkForSigning()` is called, `(*privKey).isPermitted()` always passes the `username` it receives (`&authRequest.User`) to `(*privKey).permittedByDestConstraints()`.

Recommendation

When receiving a forwarded user authentication request, do not compare the username in the authentication request to usernames in destination constraints, except when validating the last hop of the connection.

Location

- `(*privKey).isPermitted` in *ssh/agent/keyring.go*

Retest Results

2026-04-17 – Fixed

The project team reviewed commit `23bccd1`, which updates the `isPermitted()` function to only include the username when calling `permittedByDestConstraints()` when it is the last hop. Note that this finding affected code that was not yet part of the upstream repository.

In summary, this finding is considered *Fixed*.

Low

Insufficient Validation When Parsing Known Hosts

Overall Risk	Low	**Finding ID**	NCC-E029268-CAL
Impact	Low	**Category**	Data Validation
Exploitability	None	**Status**	Fixed

Impact

Incorrectly-formed *known_hosts* entries will be accepted by the implementation, contrary to OpenSSH's behavior.

Description

The *ssh* package and the *knownhosts* package both provide utilities for parsing *known_hosts* files or their contents: `ParseKnownHosts()` in *ssh*, which parses a single entry, and `New()` in *knownhosts*, which accepts a list of files and calls `(*hostKeyDB).Read()` on each of them.

Entries in a *known_hosts* file have the following format:

```
[marker] hostnames key-type key [comment]
```

where the `marker` and `comment` are optional.

This finding documents several instances of overly permissive parsing in one or both of *ssh* and *knownhosts*, or of divergences in behavior between the implementation and OpenSSH.

- **Unicode handling.** The `bytes.TrimSpace()` function (from Go's standard library) is used in various places (*ssh/keys.go* lines 105,123, 157, 218; *ssh/knownhosts/knownhosts.go* lines 176, 385) to remove leading and trailing white space from a string. This function removes all Unicode code points of the "white space" category, whereas OpenSSH would only consider ASCII space (0x20) and tabs (0x09) as white space.
- **Mismatched key types.** The function `ParseKnownHosts()` in *ssh/keys.go* intentionally ignores the `key-type` field since

```
185     // keyFields[1] contains the key type (e.g. "ssh-rsa").
186     // However, that information is duplicated inside the
187     // base64-encoded key and so is ignored here.
```

Figure 25: Excerpt from ssh/keys.go

In the *knownhosts* package, `parseLine()` also ignores the `key-type`:

```
190   // ignore the keytype as it's in the key blob anyway.
191   _, line = nextWord(line)
```

Figure 26: Excerpt from ssh/knownhosts/knownhosts.go

However, in OpenSSH (*sshkey.c*, function `sshkey_read()`), the stated key type *is* verified to match the type of the parsed key:

```
1341   /* ensure type of blob matches type at start of line */
1342   if (k->type != type) {
1343     sshkey_free(k);
1344     return SSH_ERR_KEY_TYPE_MISMATCH;
1345   }
```

Figure 27: Excerpt from sshkey.c

- **Multiple markers.** The function `ParseKnownHosts()` in *ssh/keys.go* does not check whether there is more than one marker present:

```
176     // keyFields[0] is either "@cert-authority", "@revoked" or a comma separated
177     // list of hosts
178     marker := ""
179     if keyFields[0][0] == '@' {
180       marker = string(keyFields[0][1:])
181       keyFields = keyFields[1:]
182     }
183
184     hosts := string(keyFields[0])
```

Figure 28: Excerpt from ssh/keys.go

For example, the function would not return an error if given the following string:

`@cert-authority @revoked xyz AAAAA....`

According to the `sshd(8)` manual page (the reference cited in *keys.go*), "[o]nly one marker should be used on a key line."

In the *knownhosts* package, the similar function `parseLine()` (*ssh/knownhosts/knownhosts.go*) also does not check for multiple markers, and the same example above would also successfully parse `@revoked` as the host and `xyz` as the (discarded) key type.

```
179 func parseLine(line []byte) (marker, host string, key ssh.PublicKey, err error) {
180   if w, next := nextWord(line); w == markerCert || w == markerRevoked {
181     marker = w
182     line = next
183   }
184
185   host, line = nextWord(line)
186   if len(line) == 0 {
187     return "", "", nil, errors.New("knownhosts: missing host pattern")
188   }
```

Figure 29: Excerpt from ssh/knownhosts/knownhosts.go

In OpenSSH, the `check_markers()` function (from file *hostfile.c*) rejects inputs that have more than one marker:

```
186 while (*cp == '@') {
187   /* Only one marker is allowed */
188   if (ret != MRK_NONE)
189     return MRK_ERROR;
```

Figure 30: Excerpt from hostfile.c

- **Unknown marker values.** The Go code snippet above also show some behavior differences related to the validation of the value of the marker (if present). The `ParseKnownHosts()` function (in *ssh/keys.go*) does not check what exactly the marker is; it checks only whether its first character is "`@`". The marker value is returned (this is the first of the returned values) but it is up to the caller to check that its value is correct. By contrast, in the *knownhosts* package, the function `parseLine()` (*ssh/knownhosts/knownhosts.go*) *does* perform this check; it verifies that the marker equals either of the two hard-coded strings `markerCert` or `markerRevoked`.

  In OpenSSH, the `check_markers()` function (from file *hostfile.c*) ensures that the marker is one of two known values, similarly to `parseLine()`:

```
199     if (strcmp(marker, CA_MARKER) == 0)
200       ret = MRK_CA;
201     else if (strcmp(marker, REVOKE_MARKER) == 0)
202       ret = MRK_REVOKE;
203     else
204       return MRK_ERROR;
```

Figure 31: Excerpt from hostfile.c

- **Ignored Lines:** The `ParseKnownHosts()` function (*ssh/keys.go*, lines 163-167) locates in each line (which is not a comment) the first space or tab character; if there is no such character, the line is ignored:

```
163         i := bytes.IndexAny(in, " \t")
164         if i == -1 {
165             in = rest
166             continue
167         }
```

Figure 32: Excerpt from ssh/keys.go

  This means that if the line consists of a single word, without any space (not counting the leading and trailing spaces), then it acts like a comment line, whereas OpenSSH would have reported a syntax error on such a line.

- **Constrained Comments:** The `ParseKnownHosts()` function (*ssh/keys.go*, lines 171-174) splits each line (which is not a comment) into fields, separated by white space:

```
171         keyFields := bytes.Fields(in)
172         if len(keyFields) < 3 || len(keyFields) > 5 {
173             return "", nil, nil, "", nil, errors.New("ssh: invalid entry in known_hosts
                ↳ data")
174         }
```

Figure 33: Excerpt from ssh/keys.go

  Nominally, the line should contain 3 or 4 significant fields (depending on whether an optional marker is used or not), but the rest of the line beyond the last significant field is a comment that may contain arbitrary text. Since the implementation enforces a limit of 5 fields for the complete line, comments consisting of more than one or two words will lead to an error; for instance, a comment such as "`This is Bob's key`" would trigger a parsing failure, whereas OpenSSH would have accepted that line.

Recommendation

Add the extra checks to match OpenSSH's behavior when parsing known hosts.

Location

- *ssh/knownhosts/knownhosts.go*
- *ssh/keys.go*

Retest Results

2026-04-17 – Fixed

The project team reviewed several change requests included in https://go-review.googlesource.com/c/crypto/+/782420/1:

- https://go-review.googlesource.com/c/crypto/+/782426/3 replaces the `TrimSpace()` function with a local `trimSpace()` function that only trims the correct whitespace characters.
- https://go-review.googlesource.com/c/crypto/+/782427/3 adds a check for mismatched key types.
- https://go-review.googlesource.com/c/crypto/+/782428/4 adds a check for multiple markers on the same line, and for unknown markers.
- https://go-review.googlesource.com/c/crypto/+/782429/3 adds a regression test ensuring single-word lines are rejected
- https://go-review.googlesource.com/c/crypto/+/782430/3 adds a regression test ensuring multi-word comments are accepted.

As each concern raised within this finding is addressed by one of these CRs, this finding is considered *Fixed*.

Low

Suboptimal Dynamic Diffie-Hellman Modulus Selection

Overall Risk	Low	**Finding ID**	NCC-E029268-R6B
Impact	Low	**Category**	Denial of Service
Exploitability	Low	**Status**	Fixed

Impact

When using a dynamically selected modulus for a Diffie-Hellman key exchange, the client and server settle on a group larger than what they would both have preferred, making the overall CPU usage eight times greater than it should be on both systems.

Description

The SSH protocol supports key exchange methods using Diffie-Hellman over multiplicative subgroups. Some of these key exchange methods use of one of a few fixed standard moduli (the "Oakley groups", originally defined for use with IPSec/IKE), but others let the server select a potentially custom modulus, based on supported sizes advertised by the client. The "`diffie-hellman-group-exchange-sha256`" key exchange method, specified in RFC 4419, is of that kind (with SHA-256 as companion hash function), and it is part of the set of key exchange methods supported by default by the SSH implementation.

With that method, the client first sends its minimum, maximum and preferred modulus sizes:

```
      byte    SSH_MSG_KEY_DH_GEX_REQUEST
      uint32  min, minimal size in bits of an acceptable group
      uint32  n, preferred size in bits of the group the server will send
      uint32  max, maximal size in bits of an acceptable group
```

The server uses that information to select a modulus size that both the client and server support.

In the SSH implementation, this process is handled by code in *ssh/kex.go*. The client always advertises support of all sizes from 2048 to 8192 bits, but 2048 bits is the preferred size; this is not configurable:

```
599 const (
600     dhGroupExchangeMinimumBits   = 2048
601     dhGroupExchangePreferredBits = 2048
602     dhGroupExchangeMaximumBits   = 8192
603 )
```

Figure 34: Excerpt from ssh/kex.go

As a server, the SSH implementation elects to use one of the Oakley groups 14, 15 or 16 (of sizes 2048, 3072 and 4096 bits, respectively); however, it systematically uses the largest that the client support, and disregards the "preferred" size indication:

```
722     // We hardcode sending Oakley Group 14 (2048 bits), Oakley Group 15 (3072
723     // bits) or Oakley Group 16 (4096 bits), based on the requested max size.
724     if kexDHGexRequest.MaxBits < 3072 {
725         p, _ = new(big.Int).SetString(oakleyGroup14, 16)
726     } else if kexDHGexRequest.MaxBits < 4096 {
```

```
727         p, _ = new(big.Int).SetString(oakleyGroup15, 16)
728     } else {
729         p, _ = new(big.Int).SetString(oakleyGroup16, 16)
730     }
```

Figure 35: Excerpt from ssh/kex.go

The main consequence is that while the SSH implementation claims (as a client) that it *prefers* a 2048-bit modulus, it actually selects (as a server) a twice larger modulus (4096 bits), which contradicts its own preferences, and also raises the CPU cost induced by the key exchange by a factor of about 8 (since the exponents are chosen to be as large of the modulus, inducing a typically cubic cost in the modulus size).

This behavior diverges from the implementation own's preferences, but also from OpenSSH behavior. In OpenSSH, the `choose_dh()` function on the server side scans all locally known DH groups, and keeps only the ones that have a size acceptable to the client; among those, it selects the smallest group that is at least as large as the client's preferred size, or, if there is none, the largest group that is acceptable to the client (but, in that case, still smaller than the client's preference). With the values advertised by the Go client (acceptable sizes 2048 to 8192, 2048 is preferred) and with the same set of server-known DH groups (of sizes 2048, 3072 and 4096 bits), the OpenSSH server would select the 2048-bit group, matching the client's preference.

Recommendation

The server should apply the same selection method as OpenSSH and follow the client's preferred size when possible.

Location

ssh/kex.go, lines 722-730

Retest Results

2026-04-17 – Fixed

The project team reviewed https://go-review.googlesource.com/c/crypto/+/782424/3 which updates the selection method in `(gex *dhGEXSHA) Server()` to use `chooseDH()`, aligning it with OpenSSH as recommended.

Info

SSH Agent Keyring Ignores `Confirm-Before-Use` Constraint

Overall Risk	Informational	**Finding ID**	NCC-E029268-BCX
Impact	Undetermined	**Category**	Configuration
Exploitability	Low	**Status**	Fixed

Impact

Keys that are added through *ssh/agent* with the `confirm-before-use` constraint are silently treated as unconstrained. Thus, any party that can talk to the agent can obtain signatures without the user being prompted, potentially allowing hostile forwarded connections.

Based on code comments and developer communications, it is understood that this functionality is **intentionally not supported**. As such, the Impact of this finding is rated 'Undetermined' and the Overall Risk rated as 'Informational'.

Description

The *ssh/agent* exposes the `Add()` function that enables callers to store keys with standard SSH-agent constraints. The `parseConstraints()` function implemented in the *ssh/agent/server.go* source file correctly decodes the incoming lifetime, confirm, and extension bits and will set the `AddedKey.ConfirmBeforeUse = true` when a client sends the `agentConstrainConfirm` value. The request is then handed to the in-memory `keyring.Add()` function implemented in the *ssh/agent/keyring.go* source file.

However, the `keyring.Add()` function excerpted below explicitly ignores these constraints. The highlighted comment preceding the `Add()` declaration notes that "any constraints given are ignored"; the code only copies the lifetime and destination-extension fields into the stored `privKey`.

```
// Insert adds a private key to the keyring. If a certificate
// is given, that certificate is added as public key. Note that
// any constraints given are ignored.
func (r *keyring) Add(ctx context.Context, key InputKey, session *Session) error {
  ...
  p := privKey{
    signer:  signer,
    comment: key.Comment,
  }

  if key.LifetimeSecs > 0 {
    t := time.Now().Add(time.Duration(key.LifetimeSecs) * time.Second)
    p.expire = &t
  }

  for _, ext := range key.ConstraintExtensions {
    if ext.ExtensionName == RestrictDestinationExtensionName {
      if restrictions, err :=
      ↳ParseRestrictDestinationConstraintExtension(ext.ExtensionDetails); err == nil {
        p.restrictDestinations = restrictions
      }
    }
  }
  ...
```

```
  r.keys = append(r.keys, p)

  return nil
}
```

Figure 36: Excerpt of `Add()` from ssh/agent/keyring.go

The `privKey` struct has no field to remember the confirm flag, and so `Sign()` never checks for it. Therefore a caller that explicitly required confirm-before-use when adding a key receives no indication that their requirement was ignored, as the agent stores the key, advertises it via `List()`, and signs without any interactive approval.

Note that PR `28956` circa 2016 appears to suggest the intent to honor constraints on keys but the code changes do not implement the functionality discussed above.

Recommendation

Ensure (via documentation) that package users are aware of the limitations stemming from ignored constraints.

Location

ssh/agent/keyring.go

Retest Results

2026-04-10 – Fixed

The project team reviewed *0009-ssh-agent-reject-keys-with-unsupported-confirm-const.patch*, which updates the keyring `Add()` function documentation to clarify which constraints are supported, as recommended. In particular, the `"confirm"` constraint results in an error, as do any extensions; see finding "SSH Agent Client Drops Constraint Extensions" for changes relating to extensions. Regression tests are included.

In summary, this finding is considered *Fixed*.

2026-06-02 – Fixed

The reviewed patch was merged as https://go-review.googlesource.com/c/crypto/+/778642 and disclosed as CVE-2026-39833 and GO-2026-5005.

Info

Undetected Public Key Inconsistencies

Overall Risk	Informational	**Finding ID**	NCC-E029268-K3R
Impact	Low	**Category**	Data Validation
Exploitability	None	**Status**	Fixed

Impact

Undetected inconsistencies in received public keys might conceptually help attackers evade some filtering strategies; this does not seem to be exploitable in the current implementation.

Description

Public keys exchanged on the wire for user or host authentication use a generic format which starts with a symbolic string that designates a signature algorithm; for instance, the string "`ecdsa-sha2-nistp256`" indicates that the public key that follows is of a type appropriate for using ECDSA, along with a SHA2 hash function, over NIST curve P-256. The encoded key itself follows that string. A public key appropriate for ECDSA is an encoded point defined over a given elliptic curve, which normally is one of the three NIST standard curves (P-256, P-384 and P-521); such a key is serialized as a symbolic string that designates the curve (e.g. "`nistp256`" for curve P-256) followed by the point itself (the two point coordinates). The overall format for an ECDSA public key is laid out in RFC 5656, section 3.1:

```
   The "ecdsa-sha2-*" key formats all have the following encoding:

      string   "ecdsa-sha2-[identifier]"
      byte[n]  ecc_key_blob

   The ecc_key_blob value has the following specific encoding:

      string   [identifier]
      string   Q
```

Note that the format is redundant: the used curve is specified twice, first in the algorithm string ("`ecdsa-sha2-[identifier]`"), then again in the curve identifier ("`[identifier]`" at the start of `ecc_key_blob`).

In the SSH implementation, the `ParsePublicKey()` function, defined in *ssh/keys.go* (line 293), decodes the algorithm string, then passes it to the private `parsePubKey()` function along with the rest of the key bytes. `parsePubKey()` uses the string to dispatch the decoding call to the relevant function, depending on the key type:

```
72 func parsePubKey(in []byte, algo string) (pubKey PublicKey, rest []byte, err error) {
73     switch algo {
74     case KeyAlgoRSA:
75         return parseRSA(in)
76     case InsecureKeyAlgoDSA:
77         return parseDSA(in)
78     case KeyAlgoECDSA256, KeyAlgoECDSA384, KeyAlgoECDSA521:
79         return parseECDSA(in)
80     case KeyAlgoSKECDSA256:
81         return parseSKECDSA(in)
82     case KeyAlgoED25519:
```

```
83          return parseED25519(in)
84      case KeyAlgoSKED25519:
85          return parseSKEd25519(in)
86      // etc...
```

Figure 37: Exceprt from ssh/keys.go

Note that the three plain ECDSA algorithm types (for use with NIST curves P-256, P-384 and P-521, respectively) all lead to the same call to the `parseECDSA()` function. Moreover, the algorithm type string is *not* provided to `parseECDSA()`: that function knows that it should expect an elliptic curve point, but not which curve that point should belong to. `parseECDSA()` receives the `ecc_key_blob` element (as the byte slice `in`) and obtains the curve identifier that is included in that element.

Nothing in the code verifies that the two curve specifications match. If the public key incorrectly starts with the algorithm string "`ecdsa-sha2-nistp256`" (for curve P-256) but `ecc_key_blob` really specifies a point on curve P-521 (with curve identifier "`nistp521`") then the discrepancy goes undetected; the returned `PublicKey` object will then advertise its algorithm type as "`ecdsa-sha2-nistp521`":

```
703 func (k *ecdsaPublicKey) Type() string {
704     return "ecdsa-sha2-" + k.nistID()
705 }
```

Figure 38: Excerpt from ssh/keys.go

The lack of discrepancy detection does not seem to induce any vulnerability right now; in the scenario described above, the public key is really on curve P-521, and it is the value returned by the key `Type()` method which will be checked against the negotiated authentication algorithm and signature value. A client or server cannot be induced to use in this way an elliptic curve that it did not wish to support in its configuration. Conceptually, an outer packet inspection mechanism that tries to enforce use of some specific elliptic curve could be fooled by this issue, in case it checks only the algorithm string and not the inner curve specification.

Recommendation

The `parseECDSA()` function should be provided with the received algorithm string, and verify that the curve identifier in the `ecc_key_blob` element matches that algorithm string.

Location

ssh/keys.go, line 79

Retest Results

2026-04-17 – Fixed

The project team reviewed https://go-review.googlesource.com/c/crypto/+/782425/3, which updates the `parseECDSA()` function to receive the expected algorithm string and to validate that it matches the algorithm specified in the key.

In summary, this finding is considered *Fixed*.

6 Engagement Notes

This section lists various remarks and notes that were deemed worth reporting, though none of them constitutes a security vulnerability.

Unclear References

The reference "PROTOCOL.agent" is defined twice, with different definitions:

```
18 [PROTOCOL.agent]: https://cvsweb.openbsd.org/cgi-bin/cvsweb/src/usr.bin/ssh/PROTOCOL.agent?
   ↪ rev=HEAD
```

Figure 39: ssh/doc.go

and

```
12 //[PROTOCOL.agent]: https://tools.ietf.org/html/draft-miller-ssh-agent-00
```

Figure 40: Excerpt from ssh/agent/client.go

The first definition, pointing to the OpenSSH source, describes the latest differences in OpenSSH's implementation relative to the SSH agent protocol defined in the IETF Internet Draft draft-ietf-sshm-ssh-agent. This Internet Draft is currently under development and has over 15 versions as of January 2026. The second definition points to the initial version (00) of that draft[2].

Further, in other locations in *ssh*, different versions of the Internet Draft are mentioned:

```
146   // Constraint extension identifier up to version 2 of the protocol. A
147   // backward incompatible change will be required if we want to add support
148   // for SSH_AGENT_CONSTRAIN_MAXSIGN which uses the same ID.
149   agentConstrainExtensionV00 = 3
150   // Constraint extension identifier in version 3 and later of the protocol.
151   agentConstrainExtension = 255
```

Figure 41: Excerpt from ssh/agent/client.go

Recommendations:

1. Pick one specific version of the IETF draft and/or one specific revision of OpenSSH's differences as "[PROTOCOL.agent]" and ensure the codebase implements it consistently.
2. Schedule periodic reviews of the IETF draft and OpenSSH documentation to determine whether to update this reference.
3. Ensure that any specific section numbers mentioned in the code remain consistent across version updates.

Incorrect Length Check (Harmless)

In *ssh/cipher.go*, the `writeCipherPacket()` function includes the following adjustment on the packet length to ensure that it meets the minimum allowed length:

```
589     encLength := maxUInt32(prefixLen+len(packet)+cbcMinPaddingSize, cbcMinPaddingSize)
```

Figure 42: Excerpt from ssh/cipher.go

2. The diff between version 00 and version 17 (the latest as of January 21, 2026) is available at https://author-tools.ietf.org/iddiff?url1=draft-miller-ssh-agent-00&url2=draft-miller-ssh-agent-17&difftype=--hwdiff.

The use of `maxUInt32()` is strange here since `prefixLen` and `cbcMinPaddingSize` are constants (equal to 5 and 4, respectively), so the function is comparing `9 + len(packet)` with the value 4. The length of a slice is necessarily non-negative, so the first argument must always be greater than the second, and the call to `maxUInt32()` is useless. In fact, the second argument is a typographical error, it should be `cbcMinPacketSize` (which is 16) instead of `cbcMinPaddingSize`.

In practice, this is harmless, because the next few lines round up the length to the next multiple of `effectiveBlockSize`, which is either 8 or 16; this operation implies that the length is extended to at least 16, which indeed is not lower than `cbcMinPacketSize`.

Calls to Deprecated Functions

In *ssh/kex.go* (line 277) and *ssh/keys.go* (lines 801, 907, 1635 and 1644), some standard library functions from the `crypto/elliptic` packages are used, but the standard library has explicitly declared them deprecated. These are the `Marshal()`, `Unmarshal()` and `CurveParams.ScalarBaseMult()` functions. These calls should be replaced with calls to newer APIs that are more likely to benefit from extended support in the Go standard library.

Additionally, `elliptic.Unmarshal()` can only decode elliptic curve points in uncompressed format, so that the implementation cannot accept public keys that use the compressed format. RFC 5656 indicates that "point compression MAY be used", so that compressed format support would not be out of place (`elliptic.UnmarshalCompressed()` supports the compressed format; that format can be recognized by its first byte, which is either 0x02 or 0x03, whereas an uncompressed non-infinity point starts with a byte of value 0x04). We may note that OpenSSH does not support compressed point either, so that point compression is probably never used in practice.

Ignored Lines on Connection

When an SSH connection is established, client and server should send each other "version strings", both consisting of one line of text that identifies the protocol and implementation versions. The version line starts with "`SSH-`" and ends on a newline character (0x0A). RFC 4253 Section 4.2 specifies that the maximum size of the version line is 255 bytes (including the newline character), and also that the version line may be preceded by other lines that do not start with "`SSH-`" and shall be ignored by the receiving party. The implementation slightly diverges from these rules, in function `readVersion()`:

```
347         if buf[0] == '\n' {
348             if !bytes.HasPrefix(versionString, []byte("SSH-")) {
349                 // RFC 4253 says we need to ignore all version string lines
350                 // except the one containing the SSH version (provided that
351                 // all the lines do not exceed 255 bytes in total).
352                 versionString = versionString[:0]
353                 continue
354             }
355             ok = true
356             break
357         }
```

Figure 43: Excerpt from ssh/transport.go

The implementation enforces a 255-byte limit *in total* for all received lines, including the ignored lines, whereas the text of the RFC mandates a 255-byte limit only for the version line itself, not counting the ignored lines. OpenSSH is much more lenient since it allows up to 1024 ignored lines, and also that each line (including the version line) may have size up to

8192 bytes. Since these ignored lines are meant for error messages from an underlying transport layer, the stricter limitations of the Go SSH implementation are probably not a problem in practice.

Minor Typographical Issues

- **Copy-paste error in comments.** In *ssh/messages.go*, the comment preceding the `PublicKeyUserAuthRequest` struct definition refers to RFC 4462 section 3.2 when it should instead refer to RFC 4252 section 5.

7 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.