SQL Injection Isn't Dead

Smuggling Queries at the Protocol Level

DEF CON 32 - August 10, 2024

sonar

Teaser

- In this talk, we will learn how…
  - … queries travel from app to DB
  - … attackers can inject there
  - … prevalent the problem is

sonar

SELECT * FROM speakers

```
    Name     |       Role        | Company | Team
-------------+-------------------+---------+------
Paul Gerste | Vuln Researcher |  Sonar  | R&D

(1 row)
```

SELECT * FROM speakers, companies

Name	Role	Company	Team
Paul Gerste	Vuln Researcher	Sonar	R&D

Logo	Name	Description
	Sonar	The home of Clean Code

(1 row)

sonar

Content

- The Idea
- Attacking Database Wire Protocols
- Real-World Applicability
- Future Research
- Takeaways

The Idea

Request smuggling, but for binary protocols

sonar

Prior Art

- James Kettle: HTTP Desync Attacks
  - Cause disagreement over the end of HTTP requests
- Most root causes:
  - Text parsing: 17 vs. **\t**17
  - Logical: CL vs. TE
- What about other protocols?

sonar

What About Binary Protocols?

- Binary protocols also need message boundaries
- How do they do it?
- Delimiters
  - E.g., null-terminated strings
- Length fields
  - E.g., Type-Length-Value (TLV) protocols

sonar

How To Desync Binary Protocols

- Delimiters
  - Insert delimiters into values

BLOG POST

Zimbra Email - Stealing Clear-Text Credentials via Memcache injection

Simon Scannell
VULNERABILITY RESEARCHER

June 14, 2022
DATE

https://www.sonarsource.com/blog/zimbra-mail-stealing-clear-text-credentials-via-memcache-injection/

How To Desync Binary Protocols

- Delimiters
  - Insert delimiters into values
- Length fields
  - 🤔
  - Endianness issues?
  - Overflows?

BLOG POST

Zimbra Email - Stealing Clear-Text Credentials via Memcache injection

Simon Scannell
VULNERABILITY RESEARCHER

June 14, 2022
DATE

https://www.sonarsource.com/blog/zimbra-mail-stealing-clear-text-credentials-via-memcache-injection/

Where Are Binary Protocols Used?

- Databases
- Caches
- Message queues
- … and many more

sonar

Attacking Database Wire Protocols

sonar

Why **Database** Wire Protocols?

- Extremely common
  - Almost every web app has a database
- Databases are high-value targets
  - Interesting data (e.g., PII)
  - Relevant data (e.g., for authentication)
- Guaranteed user input
  - Most queries contain some user input

sonar

High-Level Protocol Comparison

- PostgreSQL
- MySQL
- Redis
- MongoDB

High-Level Protocol Comparison

- **PostgreSQL**
- MySQL
- Redis
- MongoDB

Type	Length				Value…
'Q'	00	00	00	17	"SELECT …"

sonar

High-Level Protocol Comparison

- PostgreSQL
- **MySQL**
- Redis
- MongoDB

Length			Sequence	Value…
00	00	17	00	"SELECT …"

sonar

High-Level Protocol Comparison

- PostgreSQL
- MySQL
- **Redis**
- MongoDB

Type	Length	Delimiter	Value…	Delimiter
'+'	"17"	\r\n	"SELECT …"	\r\n

sonar

High-Level Protocol Comparison

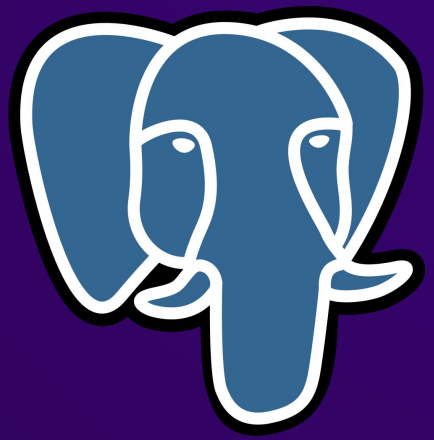
- PostgreSQL
- MySQL
- Redis
- **MongoDB**

messageLength				requestID				responseTo			
17	00	00	00	00	00	00	00	00	00	00	00
opCode				value							
DD	07	00	00	…							

sonar

Case Study:

PostgreSQL



sonar

PostgreSQL Wire Protocol

Type	Length				Value…
'Q'	00	00	00	17	"SELECT …"

- Type: 1-char identifier
- Length: 4-byte integer
- Value

sonar

PostgreSQL Wire Protocol

Type	Length				Value…
'Q'	00	00	00	17	"SELECT …"

- Type: 1-char identifier
- Length: 4-byte integer
- Value

Max value: $2^{32}-1$

sonar

PostgreSQL Wire Protocol

Type	Length				Value…
'Q'	00	00	00	17	"SELECT …"

- Type: 1-char identifier
- Length: 4-byte integer
- Value

Max value: $2^{32}-1$

The Bug

```go
func (src *Bind) Encode(dst []byte) []byte {
    dst = append(dst, 'B')
    sp := len(dst)
    // …
    pgio.SetInt32(dst[sp:], int32(len(dst[sp:])))
    return dst
}
```

The Bug

```go
func (src *Bind) Encode(dst []byte) []byte {
    dst = append(dst, 'B')
    sp := len(dst)
    // …
    pgio.SetInt32(dst[sp:], int32(len(dst[sp:])))
    return dst
}
```

Write message type

sonar

The Bug

```go
func (src *Bind) Encode(dst []byte) []byte {
    dst = append(dst, 'B')
    sp := len(dst)
    // …
    pgio.SetInt32(dst[sp:], int32(len(dst[sp:])))
    return dst
}
```

Save size offset

sonar

The Bug

```go
func (src *Bind) Encode(dst []byte) []byte {
    dst = append(dst, 'B')
    sp := len(dst)
    // …
    pgio.SetInt32(dst[sp:], int32(len(dst[sp:])))
    return dst
}
```

Write size

sonar

The Bug

```
func (src *Bind) Encode(dst []byte) []byte {
    dst = append(dst, 'B')
    sp := len(dst)
    // …
    pgio.SetInt32(dst[sp:], int32(len(dst[sp:])))
    return dst
}
```

The message buffer

sonar

The Bug

```go
func (src *Bind) Encode(dst []byte) []byte {
    dst = append(dst, 'B')
    sp := len(dst)
    // …
    pgio.SetInt32(dst[sp:], int32(len(dst[sp:])))
    return dst
}
```

Buffer length (int)

sonar

The Bug

```go
func (src *Bind) Encode(dst []byte) []byte {
    dst = append(dst, 'B')
    sp := len(dst)
    // …
    pgio.SetInt32(dst[sp:], int32(len(dst[sp:])))
    return dst
}
```

Truncate to int32

sonar

Message Size Overflow

Message 1						Message 2			
Type	Length				Value	Type	Length		
'Q'	00	00	00	08	"AAAA"	'Q'	00	00	

Size: 8 = 0x00000008

4 bytes length + 4 bytes data

Payload: "A" * 4

Message Size Overflow

Message 1						Message 2			
Type	Length				Value	Type	Length		
'Q'	FF	FF	FF	FF	"AAAA…"	'Q'	00	00	

Size: $2^{32}-1$ = 0xFFFFFFFF

4 bytes length + $2^{32}-5$ bytes data

Payload: "A" * (2**32 - 5)

Message Size Overflow

Message 1						?			
Type	Length				Value	?	?		
'Q'	00	00	00	04	""	'A'	'A'	'A'	

Size: $2^{32}+4$ = 0x100000004

4 bytes length + 2^{32} bytes data

Payload: "A" * (2**32)

sonar

Message Size Overflow

Message 1						Injected Message			
Type	Length				Value	Type	Length		
'Q'	00	00	00	04	""	'Q'	00	00	

Size: $2^{32}+4$ = 0x100000004

4 bytes length + 2^{32} bytes data

Payload: fakeMsg + "A" * (2**32 - len(fakeMsg))

Crafting a Payload

- Simple payload is easy
  - `"INSERT INTO admins VALUES … --".ljust(2**32, "A")`
- But depends on the query
  - Where is the injection point?
  - How long is the query?
- Can we make it more reliable?
  - Yes!

sonar

Crafting a Payload

- Idea: NOP sled
  - Connection stays open on non-fatal errors
  - Spam a lot of small messages (5 bytes each)
  - Hit start of a message → success
  - Hit something else → connection closed
- Success after ≤5 attempts!
  - 20% chance of success
  - Attack is repeatable, just change the offset

sonar

Crafting a Payload

- Can we make it even better?
- NOP sled v2!
  - Overlapping pattern instead of tiny messages
  - Each byte should be a valid message start
- Constraints:
  - Max message size: `0x3fffffff` → first size byte cannot be `>0x3f`
  - No valid message type `≤0x3f` :/
- Solution: Each **2nd** byte should be a valid message start
  - Hit a valid type byte → success
  - Hit something else → connection closed
- Success after ≤2 attempts!
  - 50% chance of success
  - Attack is repeatable, just change the offset

Vulnerable Libraries

- Vulnerable:
  - Go: pgx, go-pg, pgdriver, [redacted]
  - C#: Npgsql
- Unexploitable:
  - Java: pgjdbc-ng, r2dbc-postgresql
  - JS: pg, pogi, postgres, @vercel/postgres

sonar

Exploitable Applications

- Any use one of the libraries is potentially exploitable
- Requirement: smuggle 4GB into a query
- Confirmed examples:
  - Mattermost (when file upload limit ≥4GB)
  - Harbor!

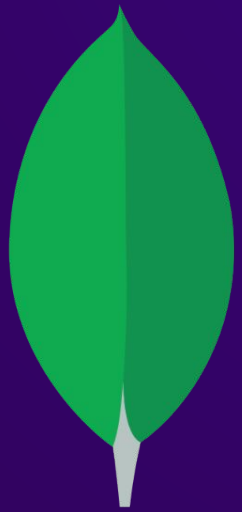
sonar

Demo: Harbor

- Container registry
- Used by VMWare Tanzu K8s
- Default configuration was vulnerable
- No authentication required

Case Study:

MongoDB



sonar

MongoDB Wire Protocol

messageLength				requestID				responseTo			
17	00	00	00	00	00	00	00	00	00	00	00
opCode				value							
DD	07	00	00	…							

- Length field in the header
- Queries are BSON documents
  - Hierarchical objects
  - Serialized to TLV sections

The Bug

```rust
async fn write_to<T: AsyncWrite + Send + Unpin>(&self, mut writer: T) -> Result<()> {
    let sections = self.get_sections_bytes();
    let total_length = Header::LENGTH
        + std::mem::size_of::<u32>()
        + sections.len()
        + /* ... */;
    let header = Header {
        length: total_length as i32,
        // ...
    };
    header.write_to(&mut writer).await?;
    writer.write_u32_le(self.flags.bits()).await?;
    writer.write_all(&sections).await?;
    // ...
}
```

The Bug

```rust
async fn write_to<T: AsyncWrite + Send + Unpin>(&self, mut writer: T) -> Result<()> {
    let sections = self.get_sections_bytes();
    let total_length = Header::LENGTH
        + std::mem::size_of::<u32>()
        + sections.len()
        + /* ... */;
    let header = Header {
        length: total_length as i32,
        // ...
    };
    header.write_to(&mut writer).await?;
    writer.write_u32_le(self.flags.bits()).await?;
    writer.write_all(&sections).await?;
    // ...
}
```

Get content bytes

The Bug

```rust
async fn write_to<T: AsyncWrite + Send + Unpin>(&self, mut writer: T) -> Result<()> {
    let sections = self.get_sections_bytes();
    let total_length = Header::LENGTH
        + std::mem::size_of::<u32>()
        + sections.len()
        + /* ... */;
    let header = Header {
        length: total_length as i32,
        // ...
    };
    header.write_to(&mut writer).await?;
    writer.write_u32_le(self.flags.bits()).await?;
    writer.write_all(&sections).await?;
    // ...
}
```

Calculate message size (usize)

The Bug

```rust
async fn write_to<T: AsyncWrite + Send + Unpin>(&self, mut writer: T) -> Result<()> {
    let sections = self.get_sections_bytes();
    let total_length = Header::LENGTH
        + std::mem::size_of::<u32>()
        + sections.len()
        + /* ... */;
    let header = Header {
        length: total_length as i32,
        // ...
    };
    header.write_to(&mut writer).await?;
    writer.write_u32_le(self.flags.bits()).await?;
    writer.write_all(&sections).await?;
    // ...
}
```

Truncate to i32

Crafting a Payload

- Must avoid bad bytes
  - Payload must be valid UTF-8
- Problem:
  - Message type (`dd 07`) is already invalid
  - Size fields can become invalid
- Solution:
  - Use metadata to create those bytes!

sonar

Demo: MongoDB Rust Driver

- The official MongoDB driver in Rust
- 3M downloads

sonar

Real-World Applicability

sonar

Constraints

- The elephant in the room:
  - Are 4GB of data realistic?
- Aren't apps limiting input sizes?
- Can `$language` handle such big payloads?

How Web Apps Handle Large Payloads

- Aren't apps limiting input sizes?
- Common protections:
  - Default body size limits
  - Maximum JSON/form decode sizes
  - Size-limiting reverse proxies
  - ... and more

sonar

How Web Apps Handle Large Payloads

- Potential bypasses
  - Unprotected endpoints
  - Compression
  - WebSockets
  - Alternate body types
  - Incrementation

sonar

How Web Apps Handle Large Payloads

- Potential bypasses
  - **Unprotected endpoints**
  - Compression
  - WebSockets
  - Alternate body types
  - Incrementation

- Some have no default limits
- Some explicitly disable the limits
  - E.g., Harbor

sonar

How Web Apps Handle Large Payloads

- Potential bypasses
  - Unprotected endpoints
  - **Compression**
  - WebSockets
  - Alternate body types
  - Incrementation

- Some enforce size limits **before** decompression
  - E.g., Nginx

sonar

How Web Apps Handle Large Payloads

- Potential bypasses
  - Unprotected endpoints
  - Compression
  - **WebSockets**
  - Alternate body types
  - Incrementation

- Compression support
- Large message size
- Many filters don't apply

sonar

How Web Apps Handle Large Payloads

- Potential bypasses
  - Unprotected endpoints
  - Compression
  - WebSockets
  - **Alternate body types**
  - Incrementation

- Some filters don't apply
- E.g., multipart forms

sonar

How Web Apps Handle Large Payloads

- Potential bypasses
  - Unprotected endpoints
  - Compression
  - WebSockets
  - Alternate body types
  - **Incrementation**

- Concat/inflate strings on the server side
- Depends on the business logic

sonar

Language Comparison

- How well do languages handle big payloads?
  - How big can strings/buffers be?
  - Are integer overflows silent?

sonar

Language Comparison: Large Payloads

Language	Max. String Size	Max. Buffer Size
Go	$> 2^{32}$	$> 2^{32}$
Java	$2^{31}-1$	$2^{31}-1$
JS	$2^{29}-24$ *	$> 2^{32}$ *
C#	$2^{31}-1$	$> 2^{32}$
Python	$> 2^{32}$	$> 2^{32}$
Rust	$> 2^{32}$	$> 2^{32}$

Only considering 64-bit versions.

* Depends on the implementation

sonar

Language Comparison: Integer Overflows

Language	Silent Addition Overflow?	Silent Serialization Overflow?
Go	Yes	No *
Java	Yes	No *
JS	No	Depends on impl.
C#	Yes	No *
Python	No	No
Rust	No	No *

* Type system prevents overflows

sonar

Real-World Applicability

- Can I send large payloads?
  - A lot of times, yes!
- Can integers silently overflow/truncate?
  - Sometimes!
- Can I exploit real-world apps with this?
  - Definitely!

sonar

Inspiration for Future Research

sonar

Safety First: No DoS Please!

Do not send large payloads to third-party systems!

Non-Invasive Detection

- White-box tests are harmless
  - Just set up your own test environment
- How to test this black-box?
  - Sending large payloads risks DoS
- More research and tools needed!
  - Can we safely detect vulnerable libraries?
  - Build tools to test this safely

sonar

Research More!

- More protocols
  - Other databases, caches, message queues
  - And more!
- Find more desync techniques
  - What about delimiters?
- More large payload methods
  - New ways to smuggle large payloads past defenses
  - Can we make the server generate the large payload?

sonar

Research More!

- All this was about **4-byte** length fields
- What about 2-byte fields?
  - Much easier to exploit (16KB vs. 4GB)
  - More to come in the future 👀

sonar

Conclusion

sonar

Takeaways

- Int overflows are relevant in memory-safe languages
- Sending large amounts of data is feasible
- If you can't hack it, just go a level deeper!

sonar

Questions?

@sonarsource

https://sonarsource.com

sonar