Securing Open Sound Control (OSC) over UDP: A Lightweight HMAC-Based Authentication Scheme for Embedded Systems

Paul V. Miller

April 28, 2026

1 Introduction

This project presents a lightweight authentication layer for transmitting OSC (Open Sound Control) packets over UDP, using HMAC-SHA256 (Hash-Based Message Authentication Code + a cryptographic hash function) and sequence-based replay protection. The approach is designed for resource-constrained embedded controllers that operate in live performance or real-time musical applications.

2 Background

2.1 UDP

UDP (User Datagram Protocol) is an important transport-layer protocol on the internet (TCP being the other). Its standards are defined in RFC 768.[1] It is designed "with a minimum of protocol mechanism", and is "transaction-oriented", meaning that it does not guarantee the "ordered reliable delivery of streams of data". We can say that UDP is part of a broader set of standards that are layered in the following way:

- Application layer (user facing) = a message of some sort
- Transport layer (UDP) = datagram
- Network layer (IP) = packet
- Link layer (Ethernet or WiFi) = frame

UDP will wrap a message in an 8 byte header which includes a source port, destination port, length and checksum. This whole thing is called the "UDP datagram". Then, IP wraps this packet with another header (usually 20 bytes at minimum) that includes information relating to routing, lifetime control, fragmentation handling and de-multiplexing. Finally, the link layer wraps everything yet again for Wifi/Ethernet transport. While these wrappers may seem somewhat redundant, in fact they serve different purposes to ensure that data is sent over the internet in an orderly way.

2.2 OSC

OSC (Open Sound Control) is a content protocol widely used in computer music and interactive systems for sending time-sensitive control messages. It is typically used in interactive and multimedia systems. In many cases, the data OSC encodes consists of some musical control information (such as frequency or amplitude) that is produced by a human interface. Many OSC software controllers are available online such as TouchOSC or Protokol. Users can install these on a so-called smartphone or tablet. These applications frequently allow the user to design a custom interface that can include virtual buttons, sliders, knobs, or other kinds of control UIs that produce data for a synthesizer or some other kind of application. A typical setup is shown below.

[1] Postel 1980.

Figure 1: TouchOSC with custom interface (image from TouchOSC's web site)

It is also possible to build one's own physical controllers out of hardware components such as sliders, buttons, potentiometers, or even more sophisticated sensors (e.g. LSM9DS1 or BNO055). A small inexpensive microcontroller such as a Raspberry Pi Pico W can then run code that formats the data into an OSC message and transmits it over a network connection.

2.3 OSC Encoding: The Details

The OSC protocol uses a hierarchical address in which each message includes a pattern that identifies its destination on the receiving end, within a tree-like namespace (somewhat like a hierarchical filesystem structure).[2] The idea behind OSC was to improve upon the 1983 MIDI standard which, though useful in many cases, is limited to 7-bit values 0-127.[3] OSC is designed to work well with patching environments such as Max/MSP and Pd, because its message tree structures can be very easily separated using the `[route]` object common in most patchers.[4]

An OSC message can mix data types such as integer, float and string:

```
/controller/move 60 0.75 "legato"
```

The command above sends a message to the address `/controller/move` with three arguments: 60, 0.75, and "legato".[5] This is the format that most users see. But OSC messages must also include a type tag string that stipulates the data types being used in the correct order. For the message above, we need:

```
,ifs
```

This particular type tag string stipulates that "the first datum is an integer, the second is a float and the third is a string". The type tag always begins with a comma.[6] To send the control message above through

[2] Wright and Freed 2002.

[3] For the official MIDI specifications, see https://midi.org/specs. The most significant bit in MIDI (the 8th bit) distinguishes status bytes from data bytes, allowing the system to differentiate between "starting a new message" (status byte) and "this byte is a continuation of the current message" (data). The designers of MIDI understood that there would probably only be 128 discrete values for pitch and volume, so they left it at that. This corresponds (very) roughly to the number of keys on the modern Steinway piano (88). The MIDI standard cannot natively send other kinds of performance data such as floating-point numbers or strings.

[4] While OSC has been widely adopted since its inception, it is not without critics. See Supper 2012.

[5] Conceptually this is something like sending parameters to a function: `move(60, 0.75, "legato")`.

[6] There are several OSC type tags in the protocol specs. To summarize:
numeric: i, f, h, d: integer, float, 8-byte int, 8-byte float
text/binary: s, b: string, blob (a binary blob can be of arbitrary length, and can transmit audio chunks, etc.)
special: T, F, N, I: these do not have any corresponding data bytes – the type tag alone carries information (True, False, Null, Infinity)
structural: t, m, r: timetag, MIDI, color in a 4-byte RGBA format

UDP, we must do the following:

1. First encode the OSC address ''`/controller/move`'' as an ASCII byte string, which is null-terminated and padded to a multiple of 4 bytes (in this particular case we have 16 bytes for the string, plus 1 for the null termination and then 3 more for padding, resulting in a final payload of 20 bytes);

2. Next, encode the type tag string ''`,ifs`'', which must begin with a comma. It is also null-terminated and padded to a multiple of 4 bytes (in this specific case we have a four-byte tag, which means we need 1 more byte for the null termination and then 3 more bytes for padding, resulting in 8 bytes);

3. Finally we can send the arguments:

   (a) Integers are 32-bit signed, and need to be big-endian (network byte order);

   (b) Floats are 32-bit IEEE 754 standard floating-point, and are also big-endian;

   (c) Strings are standard ASCII and must be null-terminated, and padded to 4 bytes, so "legato" in OSC is 8 bytes.

These standards decrease the potential ambiguity in sending OSC messages.

2.4 Threat Model: Background

Sending OSC messages usually happens over UDP, because transmission needs to be quick and low-latency. UDP does not normally engage in TCP's "three way handshake" (SYN – SYN-ACK – ACK) nor is there a sequence number involved, nor does UDP retransmit lost packets. The logic for using the UDP protocol is that in live performance, we need information quickly, and it is usually better just to drop a lost packet than to wait for it.

UDP lacks mechanisms for authentication and state. While it is possible to "spoof" TCP packets and even inject data under some circumstances, UDP doesn't have any connection state, sequence tracking or verification. So the receiver doesn't really know where a packet came from. This makes many types of attacks easier to perpetrate on UDP, than on TCP.

2.5 Threat Assessment

In a typical performance or installation that utilizes OSC over WiFi systems, an attacker has to be on the same network as the victim. For open networks, gaining this kind of access is trivial. For closed networks, it is not easy to crack a router's WiFi password (particularly if it is encoded using WPA2 or WPA3). However, musicians and artists are frequently unaware of the security vulnerabilities that routers expose. For example, routers are often shared assets that pass fluidly from person to person, with several individuals aware of the password. Router passwords are frequently shared in a cooperative environment. The sticker on the router advertising the credentials may not be removed, boldly advertising the password to any "curious" passerby who snaps a photo of the tech setup. Consequently, an attack would most likely come from an insider wanting to disrupt a performance.

If the router password is known, an attacker would still need to know the port number through which OSC data was being transmitted. In TCP and UDP, port numbers are 16 bits long. Certain ports are more frequently used for OSC transmission because of their prevalence in online tutorials: for example, most Max/MSP tutorials use ports 7000, 8000 or 9000.[7] Although port scanning is easy for TCP services discovery, it is more difficult on UDP. Although UDP services may not respond to invalid probes, it remains feasible to scan all 65,536 ports in a short period of time; a single successful OSC injection would tell the attacker that they found the right port.

Distinct from but related to message injection, *flooding* is a second problem that frequently occurs even when legitimately sending OSC control codes. Too many messages arriving at the same time can overload the ability to process them all, causing sequencing and timing issues in audio applications. Message flooding could occur if an attacker captured OSC packets and replayed them, overwhelming the victim.

[7] Another widely-used synthesis environment – SuperCollider – simply defaults to port 57120.

There are no native security features for sending OSC messages over UDP apart from the general features of WiFi routers. The protocol operates entirely on a trust model. Most artists using these systems are not trained in security measures. I have personally witnessed radically insecure setups at electronic music performances.

Breach of this trust model can easily result in a catastrophic attack. Injecting OSC messages can grind a performance completely to a halt. If the unfortunate performers were to restart, the security issues would be unresolved and the attack could resume unabated. The impact on a live performance could be enormous, potentially resulting in disruption, significant reputational harm, and audience dissatisfaction.

In summary: a *very realistic threat* could come from an attacker who already has another device on the WiFi router, and knows the port. A *moderately realistic threat* could come from someone who is on the WiFi router but brute-forces the port number. A *less realistic threat* would be an external attacker who cracks WiFi, then scans ports.

2.6 Previous Work in Securing OSC over UDP

There is no built-in security model for OSC transmission: no authentication, no integrity protection, and no encryption. No widely adopted technologies are commonly used in practice. That having been said, there are some precedents for this project. Notably, they all sit above OSC itself, focusing on securing the transport or network layers instead of the application layer.

2.6.1 WebSocket and TLS-based OSC Bridges

The aforementioned TouchOSC application supports sending OSC over WebSockets (WSS), which uses TLS/SSL encryption and integrity. WSS encrypts all the data that's transmitted, and maintains a single open, long-lived TCP connection thereby eliminating the need for repeated handshakes. It operates on port 443, and lies on the transport layer, rather than within the OSC protocol itself. Other applications that appear to use this approach include Slack and Discord.

2.6.2 liblio (OSC library) with TCP

The `liblo` library (https://liblo.sourceforge.net) supports OSC over UDP and TCP. When combined with TLS for security, this unfortunately departs from OSC's low-latency UDP implementation.

2.6.3 DTLS

This standard – outlined in RFC 6347[8] – provides "communications privacy for datagram protocols". It too is based on the TLS protocol, however it is not typically used with OSC because it has handshake overhead, and latency is not so good. So it is not very useful for musical performance.

2.6.4 SRTP

This standard is outlined in RFC 3711.[9] The "Secure Real-Time Transport Protocol" is quite close in spirit to my project in that it adds authentication via HMAC, replay protection and optional encryption. It specifies RTP ("Real-Time Transport Protocol") which apparently sits over UDP and below the application logic. While it doesn't guarantee packet delivery, it does add some structures that would be useful for real-time media communication, such as sequence numbers, timestamps, payload type tagging, and sender identification. However RTP by itself does not include authentication, encryption or integrity protection. SRTP – the secure varient of RTP – rectifies these deficiencies.

2.6.5 Assessment

As far as I understand it, no existing technique for sending OSC over networks operates on the application layer. Consequently the approach outlined in this paper is a promising foray into a field that still seems somewhat open to experimentation and development.

[8] Rescorla and Modadugu 2012
[9] Baugher et al. 2004

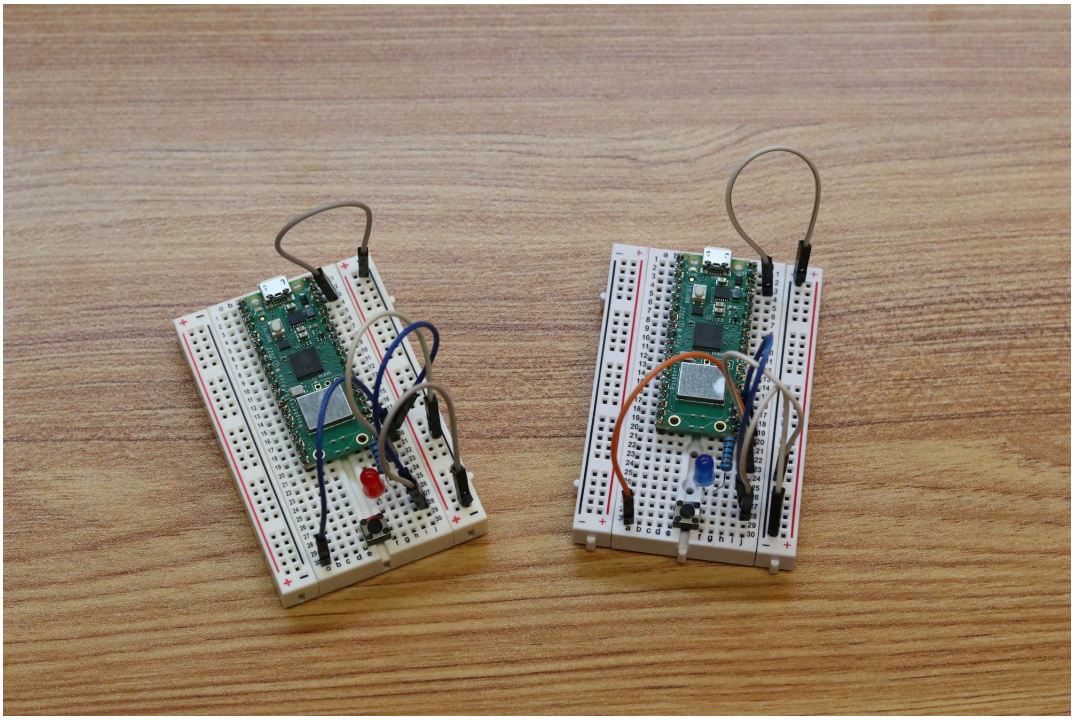


Figure 2: Unsecured Raspberry Pi Pico W builds: attacker (left, with red LED) and victim (right, with blue LED)

3 Materials and Methods

3.1 Materials

In this project I used four Raspberry Pi Pico W boards and standard, low-cost electronics components including half- and full-sized breadboards, jumper cables (Dupont style), 6×6mm buttons, a 10kΩ potentiometer, and simple LEDs for status indications. A dedicated WiFi router was necessary, since DuqNet does not allow ad-hoc UDP. I built two pairs of victim/attacker setups, to demonstrate unsecured and secured models. The code on the Pico boards is in MicroPython, a lightweight version of Python that runs on embedded systems. On the Mac we simply run Python 3. The OSC messages were received on a Mac M1 or M4 running Max 9.x. The four builds are shown below:

3.2 Methods

3.2.1 Packet Structure

The packet consisted of a header, payload and HMAC-SHA256 tag. Specifically, packets were structured in the following manner:

1. Header: 12 bytes, `>BBIIH` formatted string for `struct.pack()` and `struct.unpack()` in Python:
   (a) > = indicates big-endian byte order (for network transmission);
   (b) B = 1 byte = version number;
   (c) B = 1 byte = flags (unused for now, but available for some future purpose – "forward compatibility");
   (d) I = 4 bytes = device id (to allow for many devices, possible future expansion and for alignment simplicity);

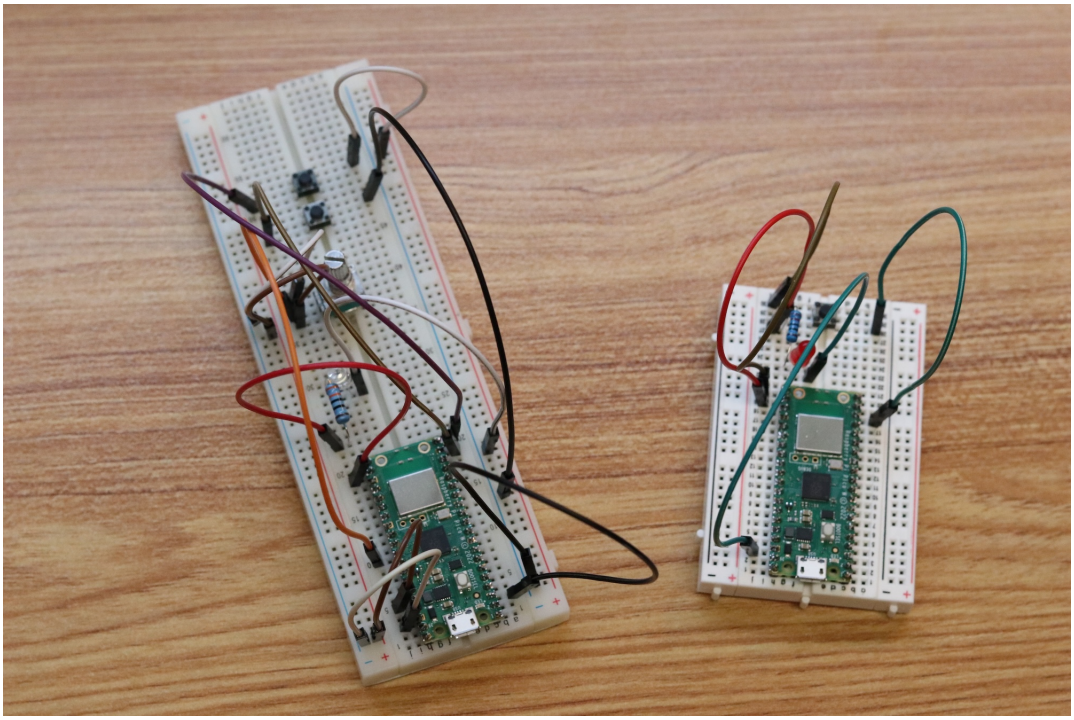


Figure 3: Secured system (left, with white LED, potentiometer and two buttons,) and attacker (right, with red LED and button).

(e) I = 4 bytes = monotonic sequence number;

(f) H = 2 bytes = payload length.

2. OSC Payload = 28 bytes

(a) The shortest control string I transmitted was `/controller/pot1`. This is 16 characters. But OSC has to be null-terminated, so this becomes 17 bytes. I padded this to 20 bytes.

(b) OSC's type tag string, e.g. ",f". This is 2 characters plus a null terminator, so 3 bytes. As per OSC specs, padded to 4 bytes.

(c) The float argument in OSC is a 32-bit float, which adds 4 bytes.

(d) Altogether this sums to 28 bytes for the entire OSC payload.

(e) Now let's examine what happens if we transmit the longest control string: `/controller/button1`. This sums to 19 characters, plus a null terminator = 20. This is already a multiple of 4, so it doesn't need padding.

(f) The type tag string for this is ",i", which is 2 characters plus the obligatory null terminator, which (when padded) again yields 4 bytes.

(g) OSC integers are 32 bits, so this is 4 bytes.

(h) The total payload size in this case is also 28 bytes.

(i) If I were to have a control string such as `/controller/button10`, I would have to increase the payload length.

3. HMAC-SHA256 tag = 16 bytes

(a) A full SHA-256 hash is 32 bytes, but I truncated it to 16 (which still gives 2^{128} possible tags, providing a good security guarantee under the threat model outlined earlier);

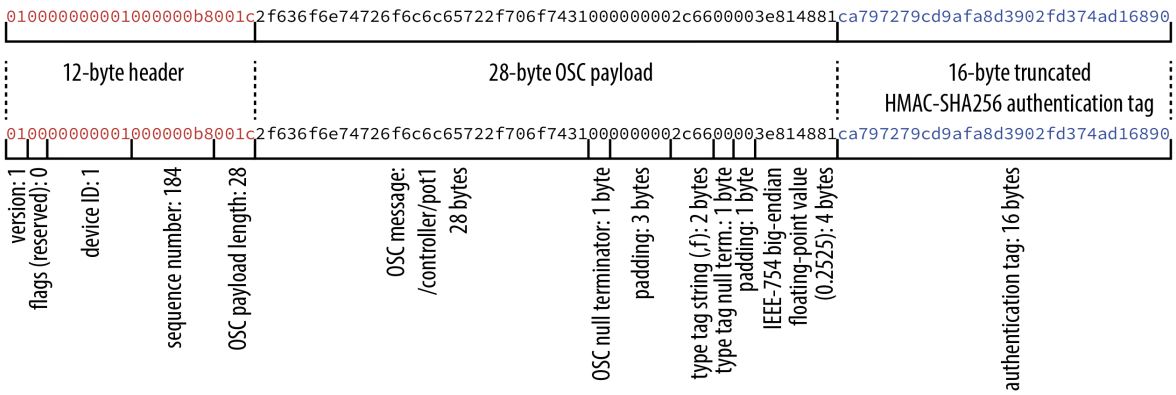


Figure 4: Secure Packet Design: 56 bytes

(b) The hash was truncated to 16 bytes to save overhead and improve latency, which is important in this project;

(c) The keyed hash was computed over the header AND the payload.

Since 12 + 28 + 16 = 56, the total size of each packet in this particular implementation is always 56 bytes. The packet design is illustrated below:

3.2.2 Authentication Mechanism

As mentioned above, this is HMAC-SHA256 over the header and the OSC payload, truncated to 16 bytes. The key is NOT transmitted and is shared "out of band". Securely sharing the key is therefore the responsibility of the people implementing the system.

3.2.3 Replay Protection

This was implemented with monotonically increasing sequence numbers. The receiver gateway rejects any duplicates or out-of-order packets.

3.2.4 Implementation Details

On the secured Pico, we need the following files:

- `controller.py` (later moved to `main.py`):[10] this is the main script that connects to WiFi, reads data from the potentiometer and the two buttons, and calls `send_secure_osc()`, which in turn calls `osc_pack()` from `osc_min.py` and `build_packet()` from `secure_packet.py`. It reports results to the REPL.
- `credentials.py`: This is a short script that contains the global constants WIFI_SSID, WIFI_PASS, DEST_IP, DEST_PORT, HMAC_KEY, a DEVICE_ID, as well as GPIO assignments for the Pico W;
- `osc_min.py`: this script helps to build the secure OSC packet;
- `secure_packet.py`: this builds the packet's header, calculates the HMAC-SHA256 hash, and truncates it to 16 bytes.

On the attacker Pico, we need:

[10] This is due to the fact that MicroPython will automatically execute any file named "`main.py`" when the Pico powers up. It is therefore quite advantageous in general, to develop and test `main.py` using some alternative filename to prevent the system from accidentally executing it on powerup. When deployed, the file is simply copied or moved to `main.py`.

- `attackx.py`: this is the script that perpetrates the attack. There are three of these: `attack1.py`, `attack2.py` and `attack3.py`;
- `credentials.py`: this is where forged credentials live, as well as all the other global constants such as WIFI_SSID, etc.;
- `osc_min.py`: this is exactly the same as the secured Pico, and builds OSC packets.

The Picos send their packets to port 9000, which is what `gateway.py` listens for on the receiving end. After the packets are received from the network socket and processed, the gateway sends diagnostic messages (logs) to stdio. It then forwards legitimate OSC to localhost on port 8000 for Max/MSP to listen to.

4 Results

I made a video demonstrating the unsecured operation, as well as three attacks on the secure system. It can be viewed on YouTube at https://youtu.be/gkkkE4qxty0.

4.1 Basic Attack on Unsecured Systems

As proof-of-concept, I simulated a basic attack using two Raspberry Pi Pico Ws without any security mechanism whatsoever. The victim simply sent OSC messages containing MIDI notes every 100ms that a Max/MSP patch on the receiving end interpreted as a three-octave ascending and descending C Major arpeggio. The attacker sent random MIDI values on the same port as the victim, dramatically disrupting the arpeggio. The MIDI notes were shown on a standard musical staff as they were received and processed thanks to the Bach/Cage library for Max/MSP (https://www.bachproject.net/). The following screenshot shows the network setup on the router and the results of the attack right at the moment after it begins, on a standard musical staff in Max/MSP:

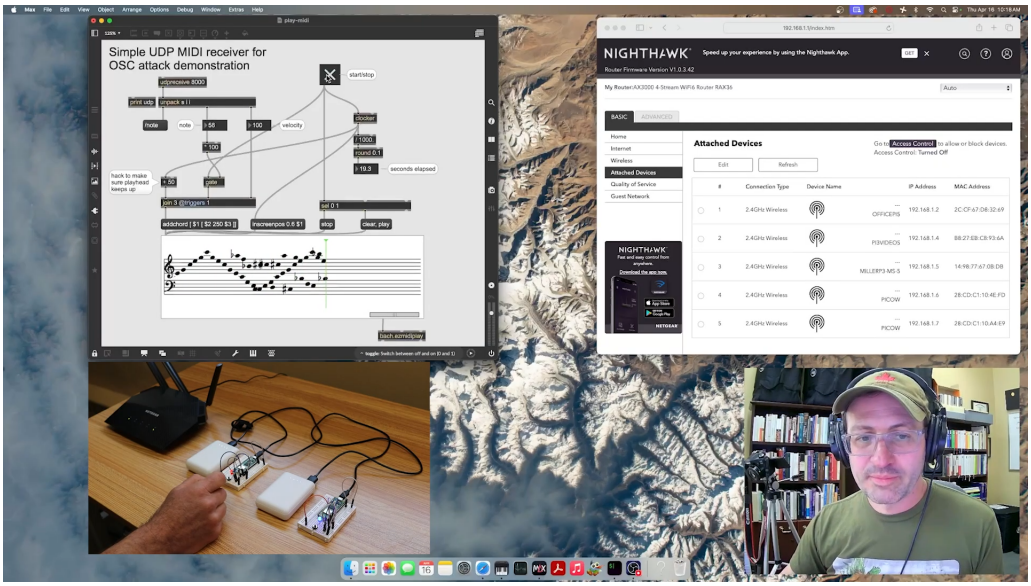


Figure 5: Unsecured OSC: screenshot from YouTube Video

4.2 Secured System Behavior

4.2.1 Secure System: Baseline Operation

The receiver-side `gateway.py` reports on its operation to stdio, whose output can obviously be redirected to a file. This log shows the normal secure operation of the system as it receives, decodes, validates and forwards secure OSC packets. The size of each packet is correctly indicated as 56 bytes:

```
Gateway is listening on 0.0.0.0:9000
Forwarding verified OSC to 127.0.0.1:8000

ACCEPT device=1 seq=200 payload_len=28 total=56  stats: ok=1 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=201 payload_len=28 total=56  stats: ok=2 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=202 payload_len=28 total=56  stats: ok=3 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=203 payload_len=28 total=56  stats: ok=4 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=204 payload_len=28 total=56  stats: ok=5 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=205 payload_len=28 total=56  stats: ok=6 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=206 payload_len=28 total=56  stats: ok=7 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=207 payload_len=28 total=56  stats: ok=8 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=208 payload_len=28 total=56  stats: ok=9 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=209 payload_len=28 total=56  stats: ok=10 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=210 payload_len=28 total=56  stats: ok=11 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=211 payload_len=28 total=56  stats: ok=12 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=212 payload_len=28 total=56  stats: ok=13 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=213 payload_len=28 total=56  stats: ok=14 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=214 payload_len=28 total=56  stats: ok=15 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=215 payload_len=28 total=56  stats: ok=16 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=216 payload_len=28 total=56  stats: ok=17 hmac=0 replay=0 len=0 bad_version_count=0
```

A diagnostic script `listen-udp.py` prints the raw packets out to stdio:

```
This is listen-udp.py - diagnostic.
Listening on UDP port 9000...
From 192.168.1.5:62692  length=56
01000000000100000b8001c2f636f6e74726f6c6c65722f706f7431000000002c6600003e814881ca797279cd9afa8d3902fd374ad16890

From 192.168.1.5:62692  length=56
01000000000100000b9001c2f636f6e74726f6c6c65722f706f7431000000002c6600003e6a8ceb40eb63f33b492914d2a21dd4e280b206

From 192.168.1.5:62692  length=56
01000000000100000ba001c2f636f6e74726f6c6c65722f706f7431000000002c6600003e528cd3f4508ce9a46ac97dee40891ea67a9388

From 192.168.1.5:62692  length=56
01000000000100000bb001c2f636f6e74726f6c6c65722f706f7431000000002c6600003e3d88be1f656a09d11dd533c20a4a98e8dc1250

From 192.168.1.5:62692  length=56
01000000000100000bc001c2f636f6e74726f6c6c65722f706f7431000000002c6600003e2348a339c306ba2f932334a657cb0a9f74e369

From 192.168.1.5:62692  length=56
01000000000100000bd001c2f636f6e74726f6c6c65722f706f7431000000002c6600003e3d08bdfca98059aafb511543666424321fc95e

From 192.168.1.5:62692  length=56
01000000000100000be001c2f636f6e74726f6c6c65722f706f7431000000002c6600003e61cce2bae795b8f168e4285def1c826163eeba

From 192.168.1.5:62692  length=56
01000000000100000bf001c2f636f6e74726f6c6c65722f706f7431000000002c6600003e770cf72da3fede65aeb01e8ebb97cec433bfdf

From 192.168.1.5:62692  length=56
01000000000100000c0001c2f636f6e74726f6c6c65722f706f7431000000002c6600003e87288720b6ea067892e33b3009da811c10cf9a

From 192.168.1.5:62692  length=56
01000000000100000c1001c2f636f6e74726f6c6c65722f706f7431000000002c6600003e924892af6260ef6fed585be2471ef9f81b73f0

From 192.168.1.5:62692  length=56
01000000000100000c2001c2f636f6e74726f6c6c65722f706f7431000000002c6600003ea60aa62df5f094921b6228bb999f98232ee499

From 192.168.1.5:62692  length=56
01000000000100000c3001c2f636f6e74726f6c6c65722f706f7431000000002c6600003e9b489b298bff8090d720167576ef81f2d3105d

From 192.168.1.5:62692  length=56
01000000000100000c4001c2f636f6e74726f6c6c65722f627574746f6e31002c690000000000014ac27aedbf58cbdec6d3c5bdc9905a17

From 192.168.1.5:62692  length=56
01000000000100000c5001c2f636f6e74726f6c6c65722f627574746f6e31002c69000000000000790bf909ec0a3e7952ebc5be5a6fcab6

From 192.168.1.5:62692  length=56
01000000000100000c6001c2f636f6e74726f6c6c65722f627574746f6e32002c690000000000001b75a56c4654b3b10e156218e990519b6

From 192.168.1.5:62692  length=56
01000000000100000c7001c2f636f6e74726f6c6c65722f627574746f6e32002c69000000000000728e809885d3aabce7513248aee120c8
```

To verify packets of potentiometer and button information were being correctly received by Max, I built a patch that mapped potentiometer value to frequency of a generic sawtooth oscillator, and each of the two buttons to a simple amplitude envelope of varying duration. The Max patch conveniently included a message that spawned a shell, executing a shell script that in turn started the `gateway.py` program. A corresponding

Max message killed `gateway.py`. This mechanism allows the Max user to start and stop the secure system from within Max itself, so the user did not have to ever venture onto the command line themselves:

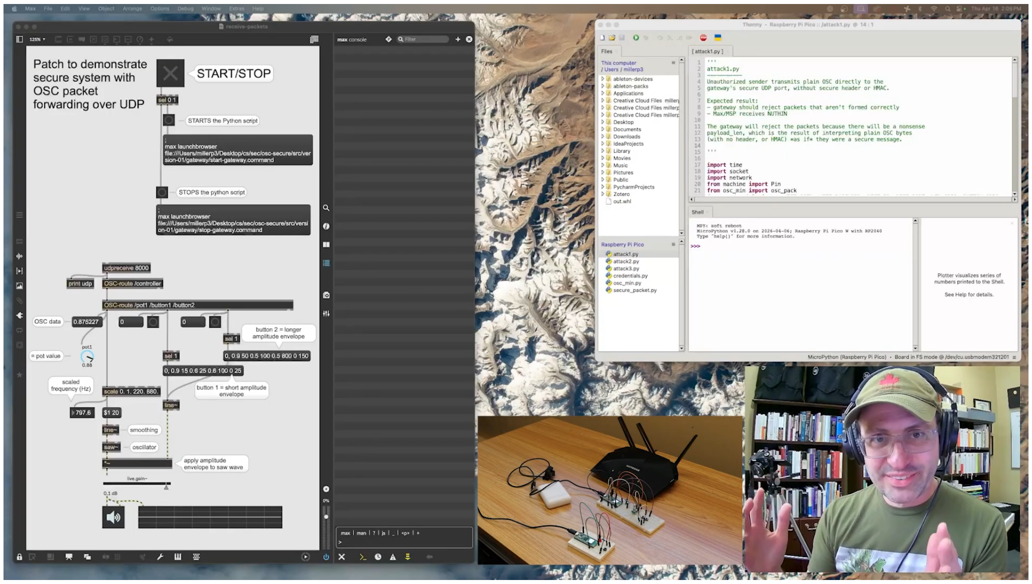


Figure 6: Secured OSC: screenshot from YouTube video

4.2.2 Attack 1: Sending plain OSC without the proper header

In the first attack, I attempted to send plain OSC packets through the secure system without a header or HMAC hash. The gateway interprets the first 12 bytes as version, flags, device_id, seq and payload_len, so it translated these plain bytes as a "garbage" payload length – which in this case worked out to 29,259:

```
Gateway is listening on 0.0.0.0:9000
Forwarding verified OSC to 127.0.0.1:8000

DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
DROP bad_length: from 192.168.1.7:63561 len=28 expected=29259
```

4.2.3 Attack 2: Correct packet structure, wrong key

Here, the attacker knows the packet design but has the incorrect shared HMAC key. The gateway responds with `DROP bad_hmac` because the hashes don't match.

```
Gateway is listening on 0.0.0.0:9000
Forwarding verified OSC to 127.0.0.1:8000

DROP bad_hmac: device=2 seq=45 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=46 from 192.168.1.7:63563
```

```
DROP bad_hmac: device=2 seq=47 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=48 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=49 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=50 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=51 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=52 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=53 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=54 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=55 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=56 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=57 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=58 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=59 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=60 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=61 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=62 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=63 from 192.168.1.7:63563
DROP bad_hmac: device=2 seq=64 from 192.168.1.7:63563
```

4.2.4 Attack 3: Replayed packets

Here, the attacker has somehow captured a legitimate packet and replayed it. In this demonstration, I recorded packets using my `listen_udp.py` diagnostic script, and simply pasted one into the attacker's Python code to be sent over UDP. A more clever hacker who positioned themselves on the network and ran some packet capture software such as Wireshark might be able to intercept packets.

Regardless of how the legitimate packet was captured, my system rejects it because it is received *out of order*. The packets are not encrypted, so this kind of attack is viable. Nevertheless the gateway rejects the out-of-order packet, while allowing legitimate packets through.

```
Gateway is listening on 0.0.0.0:9000
Forwarding verified OSC to 127.0.0.1:8000

ACCEPT device=1 seq=116 payload_len=28 total=56  stats: ok=1 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=117 payload_len=28 total=56  stats: ok=2 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=118 payload_len=28 total=56  stats: ok=3 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=119 payload_len=28 total=56  stats: ok=4 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=120 payload_len=28 total=56  stats: ok=5 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=121 payload_len=28 total=56  stats: ok=6 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=122 payload_len=28 total=56  stats: ok=7 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=123 payload_len=28 total=56  stats: ok=8 hmac=0 replay=0 len=0 bad_version_count=0
ACCEPT device=1 seq=124 payload_len=28 total=56  stats: ok=9 hmac=0 replay=0 len=0 bad_version_count=0
DROP replay: device=1 seq=26 last_seq=124
DROP replay: device=1 seq=26 last_seq=124
DROP replay: device=1 seq=26 last_seq=124
DROP replay: device=1 seq=26 last_seq=124
DROP replay: device=1 seq=26 last_seq=124
DROP replay: device=1 seq=26 last_seq=124
ACCEPT device=1 seq=125 payload_len=28 total=56  stats: ok=10 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=126 payload_len=28 total=56  stats: ok=11 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=127 payload_len=28 total=56  stats: ok=12 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=128 payload_len=28 total=56  stats: ok=13 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=129 payload_len=28 total=56  stats: ok=14 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=130 payload_len=28 total=56  stats: ok=15 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=131 payload_len=28 total=56  stats: ok=16 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=132 payload_len=28 total=56  stats: ok=17 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=133 payload_len=28 total=56  stats: ok=18 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=134 payload_len=28 total=56  stats: ok=19 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=135 payload_len=28 total=56  stats: ok=20 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=136 payload_len=28 total=56  stats: ok=21 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=137 payload_len=28 total=56  stats: ok=22 hmac=0 replay=6 len=0 bad_version_count=0
ACCEPT device=1 seq=138 payload_len=28 total=56  stats: ok=23 hmac=0 replay=6 len=0 bad_version_count=0
```

4.3 Performance

I did an analysis of latency by making a parallel branch of the code. Latency was measured in two ways: first as a round-trip timing (RTT), and second as gateway processing overhead on the Mac (receiving end). The RTT method was chosen because it is not possible to synchronize clocks on the Raspberry Pi Pico W and the Mac.

The results of analyzing 30 packets reveal much interesting information. I found that latency was almost the same for plain OSC and secure OSC. Plain RTT averaged 211.1 - 212.0ms for most samples, with one outlier at 310.8ms. Secure RTT also averaged 211.2 - 212.2ms for most samples, with one larger outlier at

512.8ms. This data allows me to conclude that the practical end-to-end RTT in the project is dominated by something other than HMAC verification – most likely, the latency in WiFi variability, network jitter, OS scheduling, or perhaps buffering delays.

A better indication of the actual cost of the security layer was gateway-side timing. Gateway-side timing is the time taken to parse, verify and forward secure OSC packets to the loopback after the Python script receives them from the OS. This includes the time it takes to:

1. extract version, flags, device_id, seq, payload_len;
2. check the length of the data to make sure it conforms to expected_len;
3. check the recovered HMAC tag to make sure it conforms to the expected tag (using `hmac.compare_digest()`...)
4. enforce replay protection;
5. and finally, forward the verified OSC string to the loopback with `forward_sock.sendto(payload, ('127.0.0.1', 8000)`.

Notably, gateway-side timing explicitly *does not* include the network transit time it takes the packet to move from the Pico to the Mac, or socket receive blocking time. The timer starts only *after* we receive a packet from the OS socket:

```
recv_sock.bind(('0.0.0.0', 9000))
data, addr = recv_sock.recvfrom(...)
```

In the plain gateway-side timing, processing time averaged about 160.8μs over 30 packets, with a range of 97.2 – 210.0μs, while in the secure gateway-side timing, processing averaged 218.6μs with a range of 146.9 – 366.5μs. So we can conclude that **the secure version adds about 58μs on average** to the gateway overhead compared to the plain OSC. This is negligible within the parameters of this project. The results are summarized in the table below.

Measurement	Range	Mean	Outlier(s)
Plain OSC: Round-Trip Time	211.1 – 212.0 ms	211.5 ms	310.8 ms
Secure OSC: Round-Trip Time	211.2 – 212.2 ms	211.8 ms	512.8 ms
Plain OSC: Gateway Processing Time	97.2 – 210.0 μs	160.8 μs	none
Secure OSC: Gateway Processing Time	146.9 – 366.5 μs	218.6 μs	366.5 μs

Table 1: Latency Times as Measured over 30 Packets

5 Discussion

This lightweight system for authenticating OSC over UDP worked well, added an insignificant amount of latency, while surviving three simulated attacks. When a user installs the Python code on the transmitting and receiving ends, the system significantly improved resistance to spoofing, injection and replay attacks.

The most significant weakness in the current system is that a longer OSC payload is not supported, because the OSC string is limited to 28 bytes. A design goal for a potential version 0.2 would be to allow for a variable-length OSC payload. Ultimately, the user should determine the length of the OSC command string, not the system. A future version would respond to whatever length OSC string the user sent into it, within reason. Because the processing time of each packet is essentially insignificant, there is considerable headroom remaining which could accommodate longer strings.

A second area in which the current system could be improved is in the replay protection function. The current implementation enforces strictly monotonic sequence numbers, rejecting any out-of-order packets. While this provides replay protection, it also means that any packet received out-of-order will automatically be discarded. While there is a good musical argument for simply discarding any older control states, it might be best to let the user decide the degree to which strict monotonic time should be enforced. A future design could employ a user-defined sliding acceptance window to tolerate slight disorder, while still preventing replay attacks.

6 Conclusion

My system provides *integrity* through HMAC, *authenticity* through a shared key, but lacks confidentiality as there is no encryption. This was a purposeful design choice, as I did not want to increase latency by introducing an encryption/decrypion layer. However, noting that the cost of the current system is so low, encryption might be worth pursuing in the future.

Although an attack on transmitting OSC information over UDP is unlikely, it is potentially catastrophic in a performance environment. My system provides a robust layer of security to defend against three possible attack vectors, significantly increasing the overall system resilience.

References

Baugher, Mark et al. (2004). *The Secure Real-time Transport Protocol (SRTP)*. RFC 3711. RFC Editor. URL: `https://www.rfc-editor.org/rfc/rfc3711`.

Bishop, Matthew (2018). *Computer Security: Art and Science*. 2nd. Boston: Addison-Wesley.

Krawczyk, Hugo, Mihir Bellare, and Ran Canetti (1997). *HMAC: Keyed-Hashing for Message Authentication*. Tech. rep. RFC 2104. IBM and UCSD; RFC 2104. URL: `https://www.rfc-editor.org/rfc/rfc2104`.

Postel, Jon (1980). *User Datagram Protocol*. Tech. rep. RFC 768. RFC Editor. URL: `https://www.rfc-editor.org/rfc/rfc768`.

Rescorla, Eric and Nagendra Modadugu (2012). *Datagram Transport Layer Security Version 1.2*. RFC 6347. RFC Editor. URL: `https://www.rfc-editor.org/rfc/rfc6347`.

Supper, Ben (Oct. 2012). *We hate MIDI. We love MIDI.* Internet. URL: `https://focusritedevelopmentteam.wordpress.com/2012/10/24/we-hate-midi-we-love-midi/`.

Wright, Matthew and Adrian Freed (2002). "Open Sound Control: A New Protocol for Communicating with Sound Synthesizers." In: *Proceedings of the International Computer Music Conference (ICMC)*.

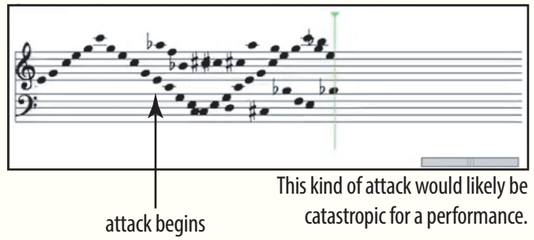
Securing OSC (Open Sound Control) over UDP

paul v. miller // computer security 2026

1.1 The Problem or Challenge

This project presents a lightweight authentication layer for transmitting OSC (Open Sound Control) packets over UDP, using HMAC-SHA256 (Hash Message Authentication Code + a cryptographic hash function) and sequence-based replay protection. It is designed for resource-constrained embedded controllers that operate in live performance or real-time musical applications.

An attack on OSC: A victim transmitting OSC code that translates to a 3-octave C major arpeggio is attacked by a second device on the same network and port, transmitting random notes.



This kind of attack would likely be catastropic for a performance.

1.2 Secure System Overview

Input from hardware sensors → **Packet encoding on Pico W** → **Send through WiFi router via UDP** → **Received on Port 9000** → **Processed by gateway and sent via OSC to localhost on port 8000** → **Received on port 8000 by Max/MSP**

These hardware sensors can be anything connected to the GPIO pins on a Raspberry Pi Pico.

The Pico encodes a secure packet and sends it over WiFi.

The packet passes through the WiFi router to the destination system

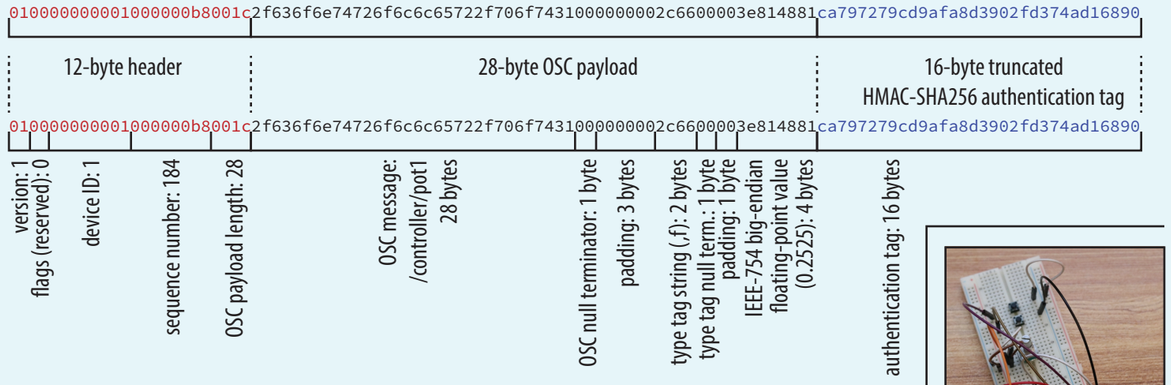
A gateway script receives the secure packet, and authenticates it.

The gateway forwards the authenticated packet to the loopback (localhost)

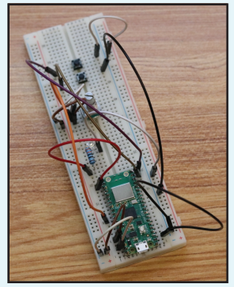
A patcher (in this case, Max/MSP) receives OSC from the loopback interface

2. Secure Packet Design

On the secure system, the Raspberry Pi Pico W runs MicroPython code that reads values from a minimal musical control system consisting of a 10kΩ potentiometer and two buttons. It creates a 12-byte header, then appends the OSC string, and finally adds a 16-byte truncated HMAC-SHA256 authentication tag. It sends this packet through UDP over WiFi to the destination IP address at port 9000. The secure packet is constructed like this:



On the receiving end, a gateway receives the packet on port 9000, authenticates it, and then forwards it to the loopback (127.0.0.1) on port 8000. From here, Max/MSP (or, for that matter, anything else on the system that is listening on port 8000) can get the authenticated OSC packet.



Secure system: Raspberry Pi Pico W with potentiometer, white status LED and two 6×6mm buttons

3. Threat Model and Attack Resilience

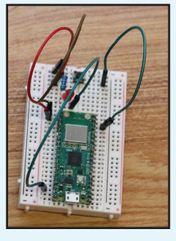
For an attack to occur, the malevolent agent must be on the same WiFi network and know the port that the victim is using. It is possible to scan all 65,536 ports in a few seconds; any successful hit will probably be heard. Also due to the prevelance of online tutorials, a few ports (7000, 8000, 9000) are very frequently used for OSC transmission over UDP. Therefore a successful attack is most likely going to happen from an insider. While it is not very likely, the results of an attack would be catastrophic, possibly leading to a total collapse of the performance and significant reputational harm. I subjected the secure system to three attacks from a separate Raspberry Pi Pico W.

3.1 First Attack

The attacker sent regular OSC packets with no header or HMAC tag. The secure system rejected these and outputted the message DROP bad_length to stdio. Bad packets were not forwarded to the loopback.

3.2. Second Attack

The attacker sent a packet with the right structure (header, payload, HMAC tag) but the HMAC tag was generated with the wrong key. The secure system rejected the bad packets and responded with the output "DROP bad_hmac" to stdio. No bad packets were forwarded to the loopback.



Attacker Raspberry Pi Pico W: One button (on/off) and red status LED

3.3 Third Attack

The attacker somehow captured an authentic packet with the correct structure and properly generated HMAC tag. In this case, I simply replayed packets that the system received, using a diagnostic script I wrote. A better hacker could potentially acquire packets by sniffing them. The secure system rejected the injected packet because its monotonic sequence number was out of order, printing the message "DROP replay" to stdio. Again, no bad packets were forwarded to the loopback.

4. Outcomes and Results

The system survived all three attacks, while providing *integrity* through HMAC and *authenticity* through a shared key. There was no *confidentiality* assurance as the system did not use any data encryption: a sacrifice that was made in order to ensure speed.

I measured system latency in two ways. First, I calculated round-trip packet time as there is no way to synchronize clocks on the Pico and the receiver. These data include WiFi transit time, OS scheduling and possible buffer delays. A more accurate measure of the latency that the system introduces is gateway processing time. This is the time it takes to process the secure OSC packets. Gateway processing time is extremely low, averaging **58 µs**:

Measurement	Range	Mean	Outliers	
Plain OSC: Round-Trip Time	211.1 – 211.5 ms	211.5 ms	310.8 ms	
Secure OSC: Round-Trip Time	211.2 – 212.2 ms	211.8 ms	512.8 ms	
Plain OSC: Gateway Processing Time	97.2 – 210.0 µs	160.8 µs	none	difference: **58 µs**
Secure OSC: Gateway Processing Time	146.9 – 366.5 µs	218.6 µs	366.5 µs	

Latency Times as Measured over 30 Packets

Future versions of the system should at minimum include more flexibility in the length of the OSC payload. Other features that may be worth pursuing are data encryption (though this could add substantially to latency), and a sliding acceptance window that would accept packets if received slightly out of order. These options should probably not be hard-wired into the system, but rather options that the user could select themselves.