SECURITY IN NETWORK CONNECTED PERFORMANCE ENVIRONMENTS

ABSTRACT

In this paper we highlight security issues created through the use of network connectivity in performance. We argue that an awareness of these issues can lead to more secure and stable software, in both a technical and a musical sense. Potential exploits that can compromise performance integrity are illustrated along with suggestions for methods alleviating such concerns.

1. WHY SECURITY?

1.1. Security a relevant concern

Security as a concern within performance practice is rarely considered a priority, as the perceived threat level from third parties is low[1]. However, trust of any apparent user input is a crucial to the successful musical performance. It is our contention that, whilst the threat from malicious users is perhaps at present minimal, software for network performance is often needlessly compromised in terms of security, leaving it open to both malicious attack, and (more realistically) failure through unintentional behaviour. In the worst cases these risks can be catastrophic. The underlying problem is implicit trust of all incoming data, which we propose should not be the *de facto* standpoint.

1.1.1. The Trusted User ?

We start by considering a non-networked scenario, where all participants are known and in which the software is written by a single composer or creator. This common compositional practice of a composer creating a performance environment/tool for use by a group of performers is inherently a relationship of common intention, if not one of complete trust.

It seems logical in this scenario to consider control data from such users to be valid by default, as user interaction, within a provided graphical or text-based interface should limit parameters to sane values. This measure militates against the performance becoming compromised, either through damage to the musical integrity of the performance from spurious values, or through potential risks to the stability of the application.

1.1.2. The Network Attached User

With the increased availability of computing resources, the emergence of enabling technologies for networking, such as Open Sound Control [9], and the practice of creating distributed environments with network connectivity [3] as carried out by the Milwaukee Laptop Orchestra [7] network performance has become more common. There are also an increasing number of networked performances in which each participant provides his/her own system, but with the facility for some or all parameters to be accessible to others on the network.

In either of these scenarios the inherent user trust of the isolated device no longer applies (as in non-networked laptop performance). Instead, we argue that the default position should be one of distrust, albeit not always due suspicion of malicious intent, but rather simply due to the unknown source or reliability of any network data.

Whereas in a non-networked situation as outlined in 1.1.1, the protection of the user interface limitations can be assumed, this is not the case in a networked performance. While the performance intention may be based around the use of a common interface or software, the lack of control over all possible networked attached devices offers the possibility for intervention by unknown users and/or unintended or inappropriate user interfaces. When a common software interface is not used, there is no guarantee that the interface will offer suitable protection against the insertion of hazardous values, intentional or otherwise.

While it is important to consider the malicious user, a more common situation may be the incorrectly configured or inexperienced user creating inappropriate values exposing security problems.

1.2. Scope

Platform stability, crucial to performance, is directly susceptible to risk through possible attack vectors within the operating system and user written code. With the exception of open-source programs, the lack of attack vectors within the underlying operating system and application internals is largely a matter of trust between vendor and client, outside of the scope of this paper[2].

Rather, this paper will focus on the security and stability risks created through use of user written code within

[1] Perhaps this is due to the lack of readily available tools or prerequisite knowledge and methods, as the desire for an unsatisfactory performance to end early has surely been experienced by most concert-goers!

[2] The transparency of the code base within an open-source project offers the ability to state that code is exploit free rather than trusting vendor assurances.

performance environments such as ChucK and Max. It should be noted that the illustrated attacks are designed to demonstrate the potential for sabotage, disruption (intentional or otherwise) or prevention of an intended performance only. The compromise of entire systems are not of concern here.

2. EXPLOITS

The proof of concept exploits illustrated here are designed to operate in realistic performance environments. A typical setup is a user system accepting udp packets on a specified port number via OSC over a WIFI network (as required for Scott Smallwood's PlorK composition 'On The Floor' [6] for example).

2.1. Severe

The severe exploits detailed in **Table 1** offer the possibility of disrupting the entire application by exploiting language features, rather than bugs. In doing so, the exploits illustrate the requirement for security conscious user programming. The languages Max and ChucK were chosen due to author familiarity rather than any perceived susceptibility. This should not be viewed as a indictment of the chosen applications rather these exploits were developed to illustrate the severity of lack of security.

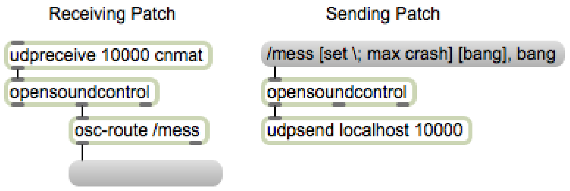
Whilst these exploits are not universal, in that they require particular code configurations at the receiving end, we contend that these configurations are not unrealistic, and also appearing innocuous, with no obvious indications of risk.

2.1.1. Max

Max allows the user to control the application directly by inserting the text *; max* into a message box, followed by the message to be sent. In **Figure 1**, the OSC bundle inserts , *; max crash* into a message box and bangs it immediately, causing Max to crash.[3].

In the example, the namespace is known, but in fact OSC wildcard matching can be used to send any namespaces allowing this exploit.

Figure 1. Max Severe Exploit



Receiving Patch

```
udpreceive 10000 cnmat
opensoundcontrol
osc-route /mess
```

Sending Patch

```
/mess [set \; max crash] [bang], bang
opensoundcontrol
udpsend localhost 10000
```

2.1.2. ChucK

The ChucK programming language [8] is built around the manipulation of time both at sample rate and also in larger more musically relevant durations. While ChucK has a primitive data duration, it is typical to communicate durational values into ChucK as either ints or float. Setting the single durational statement within a while loop structure to zero (as in **Figure 2**) will cause the VM to hang and eventually crash.

Figure 2. ChucK Severe Exploit

```
// OSC input set to OSC_Value
while(1){
    OSC_Value * 1::ms => now;
}
```

2.2. Nuisance

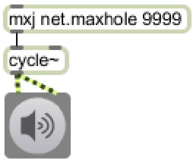
Rather than causing complete system crashes, these 'nuisance' exploits (see **Table 2**) undermine the musical integrity of the performance. Attacks of this type require far less specific coding configurations, rather relying on the insertion of unexpected parameter values. Thus, they are far easier to achieve, especially due to the human readable nature of OSC namespaces, and popularised use of udp as a transport protocol. While the outcomes may not crash the application, unintended tempo or pitch values would be a significant hindrance to a performance as would altering audio oscillator values to LFO range[4].

The methodology for a malicious user trying to identify such nuisance opportunities might be as follows;

1, examine network traffic
2. identify appropriate range of values
3. prepare code to transmit nuisance value
4. send value.

2.2.1. Examples of Nuisances - Rogue values

Figure 3. Max Nuisance Exploit



```
mxj net.maxhole 9999
cycle~
```

Figure 4. Chuck Nuisance Example

```
// OSC input set to OSC_Value
SinOsc s => dac;
while(1){
    OSC_Value => s.freq;
    100::ms => now;
}
```

In the example code (**Figure 3** and **Figure 4**), an oscillator frequency is set directly by an incoming value. The oscillator's audio output is then sent directly to the audio interface output. As this signal is to be heard, the implicit expectation is that incoming values will set be within the audio range. However, it is entirely possible to set the

[3] Internally Max attempts to dereference a null pointer.

[4] It should also be noted that an appropriately ranged value transmitted at the wrong time would also be consider a nuisance value.

Table 1. Severe Exploits

Language	textitOSC Message	*Outcome*	*Language Feature*	*Programming Requirements*
Max	;max crash	Crash	Exposed Development Tool	Insertion into message box, followed by bang
ChucK	0	VM Lockup	Control of Time	Value set as only duration within control structure

Table 2. Nuisance Exploits

Detected OSC Message	*Range*	*Nuisance Value*	*Outcome*
/freq1	400-800	0.1	Audio Outcome
/bpm	120	400	Tempo Change
/dsp	on-off	off	Audio System Off

frequency to an inappropriate value in terms of the ongoing performance or even outside of the audio range, potentially reducing the amplitude range available to valid signals, and perhaps causing damage to loud speakers or other equipment.

3. CURRENT HIGH-LEVEL NETWORK SECURITY PRACTICES

The exploits identified require network connectivity. Consequently we consider typical security counter measures in relation to both the trusted user and malicious user.

3.1. Physical Transit

The physical transit of the network connection must be compromised to allow a malicious attack or even an incorrectly configured client. A fully wired network offer a significant obstacle not in part due to the difficultly of hiding a physical intrusion in a performance environment, however the issues surrounding trusted users remain a concern.

3.1.1. WIFI Security (Social Engineering Hacks)

Typical WIFI network security offers only limited protection from malicious users. WEP is now considered compromised [4], while WPA, though cryptographically still secure, is susceptible to guessing or social engineering cracks targeted at pass phrase acquisition. Consequently, data from a wireless network should not be considered secure and the trusted user issues remain[5].

3.2. Firewall

While a correctly configured Firewall can be used to prevent network connections, client applications are designed to receive network data and thus appropriate firewall ports must be left open. However the firewall could be configured to only accept packets from approved IP clients.

3.3. Multicast

The use of multi-casting (e.g. using *mxj net.maxhole* object in Max) removes the complexity of network configuration, and is convenient for performance,. However, as all network traffic is mirrored to all network connected devices, profiling network traffic is easy. Additionally as a udp receiver accepts all inbound traffic regardless of source, a device need only send data to inject values[6].

3.4. Obscurity

One of the major features of the OSC style namespaces is the human readable and context-specific nature of the namespaces. A self identifying namespace offers opportunity to send appropriate data with an understanding of its context. Whilst namespaces could be obscured in order to increase security, this approach runs counter to the design of the protocol, and is not an appropriate solution for scenarios in which human examination of namespaces is a strong requirement, rather than simply a convenience (e.g. a network jam)[7]. Regardless of the viability of this approach in a given situation, it is arguable that examination of the network traffic will expose the purpose of specific namespaces, due to the observable range and frequency of change of parameters.

4. SUGGESTED SOLUTIONS

Having identified these security issues we now suggest some methods that offer protection from these behaviours.

4.1. Transport Layer Security

While the ease of use of udp protocol for multi-casting or receiving is attractive, the security issues are significant, consequently we would suggest using the TCP protocol as doing so would require an intruder to also spoof a connected device. [8] The use of an ssh tunnel, or VPN, within an encrypted WIFI or physical network would present a demanding network to penetrate, unlikely within the timescale of a performance.

[5] Wireless networks are also susceptible to denial of service style attacks [5] and due to the lack of physical connection, attempted intrusion of this sort is more subtle.

[6] A conceivable example of this would be a workshop participant leaving the software running through the performance

[7] The web article 'Security Through Obscurity.'Ain't What They Think It Is' [1] by Jay Beale offers a discussion regarding the appropriate use of obscurity

[8] Readers concerned about the latency implications of TCP transport are directed to [2] who suggests that the penalty under normal situations is negligible.

While transport layer security offers a degree of protection against a malicious attacker it offers no protection from nuisance value insertion.

4.2. Data Obfuscation

Obfuscating namespaces offers limited protection, however analysis of sent values will still identify possible nuisance values. It does however make interacting with network control data more difficult. While encrypting OSC traffic from within the application space is an option, an ssh tunnel is likely to be more secure and requires less maintenance by the programmer. Consequently we do not advocate security through obfuscation.

4.3. Data Sanitisation

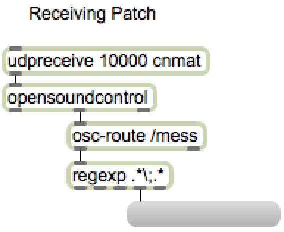
While severe code exploits are worth identifying and avoiding nuisance value insertion is always inherently possible. As discussed above ensuring incorrect values are not inserted into a data stream is a difficult task and as a consequence it should be assumed that not all values within the data stream are suitable for use. Therefore incoming data should be sanitised.

This sanitisation should be considered a two part process: firstly, filtering out inappropriate data types; secondly blocking out-of-range values. Obviously, such filtering will be context dependent informed by musical intentions. Within the performance paradigm it is also conceivable to vary the criteria for acceptable values in line with score directions.

4.3.1. Max Exploits Sanitised

It is arguable that the *; max* command should be escaped in all cases due to its severity. In **Figure 5**, the input is sanitised by disallowing any messages containing a semicolon to be send to a message box. The regexp object matches anything with a semicolon and only unmatched messages are sent on to the message box.

Figure 5. Max Severe Exploit Sanatised



```
udpreceive 10000 cnmat
opensoundcontrol
osc-route /mess
regexp .*\;.*
```

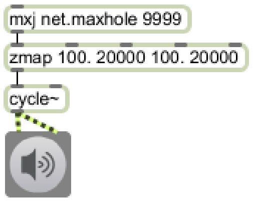
Through the use of zmap in **Figure 6** only values within the audio frequency range pass through the patcher.

4.3.2. ChucK Exploits Sanitised

The if control structure within **Figure 7** prevents a zero value being set as the loop duration.

The if control structure within **Figure 8** prevents values outside of the approved range from being used.

Figure 6. Max Nuisance Exploit Sanitised



```
mxj net.maxhole 9999
zmap 100. 20000 100. 20000
cycle~
```

Figure 7. ChucK Severe Exploit Sanitised

```
// OSC input set to OSC_Value
while(1){
    if (OSC_Value != 0){
        OSC_Value * 1::ms => now;
    }
}
```

5. CONCLUSIONS

Within this paper we have sought to illustrate the risks in network connectivity within musical performance and highlight the severity of the issues presented. Having evaluated possible security counter measures we have identified limitations to common approaches and highlighted the need for distrust amongst network environments. Finally we have demonstrated simple data sanitisation to mitigate the most severe risks and provide greater stability to network performance systems.

References

[1] J. Beale. (2000) "security through obscurity" ain't what they think it is. [Online]. Available: http://web.archive.org/web/20070202151534/http://www.bastille-linux.org/jay/obscurity-revisited.html

[2] R. B. Dannenberg, "Laptop orchestra communication using publish-subscribe and peer-to-peer strategies," Baton Rouge, 2012.

[3] G. Essl, "Automated ad hoc networking for mobile and hybrid music performance," Huddersfield, 2011.

[4] S. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the key scheduling algorithm of RC4," in *Selected Areas in Cryptography*, S. Vaudenay and A. M. Youssef, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, vol. 2259, pp. 1–24.

[5] K. Pelechrinis, M. Iliofotou, and S. Krishnamurthy, "Denial of service attacks in wireless networks: The case of jammers," *Communications Surveys Tutorials, IEEE*, vol. 13, no. 2, pp. 245 –257, quarter 2011.

[6] S. Smallwood, D. Trueman, P. R. Cook, and G. Wang, "Composing for laptop orchestra," *Computer Music Journal*, vol. 32, no. 1, pp. 9–25, Mar. 2008.

[7] G. Surges and C. Burns, "Networking infrastructure for collaborative laptop improvisation," 2008.

[8] G. Wang and P. R. Cook, "ChucK: a concurrent, on-the-fly, audio programming language," 2003.

[9] M. Wright, "Open sound control: an enabling technology for musical networking," *Organised Sound*, vol. 10, no. 3, pp. 193–200, 2005.

Figure 8. ChucK Nuisance Exploit Sanitised

```
// OSC input set to OSC_Value
SinOsc s => dac;
while(1){
if (OSC_Value > 100 && OSC_Value < 20000){
        OSC_Value => s.freq;
        100::ms => now;
    }
}
```