What You Need to Know About X11 Forwarding

X11 forwarding, `ssh -X`, is an SSH protocol that enables users to run graphical applications on a remote server and interact with them using their local display and I/O devices. It is commonly relied upon by developers for securely interacting with remote machines across wide and heterogeneous server fleets.

Here at Teleport we recently added X11 Forwarding to our list of supported SSH protocols. As we implemented it, we began to realize that despite its common usage, there are very few sources which accurately explain how X11 Forwarding works. In this blog post, I'll share some insights to answer common questions about X11 and X11 forwarding, as well as cover the security implications associated with X11 Forwarding that any user would benefit from understanding.

What is X11?

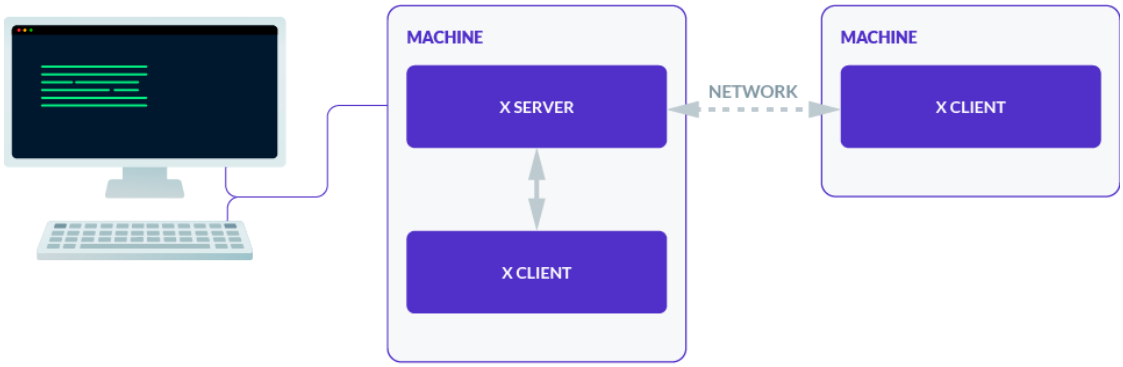
X11 refers to the 11th edition of the X Window System; an open source graphics protocol developed in the early days of the internet. It provides a basic framework for creating custom GUIs that can display graphics on both local and remote display devices. The remote capabilities of X11 were useful in the early days of the internet, where "super" computers would handle the heavy lifting for several users on separate workstations, sometimes over remote networks.

X11 was initially a pretty basic protocol, but over the past few decades it has been extended to include modern features, such as the Shared Memory Extension which drastically improves performance of X11. Newer projects like Wayland are quickly outperforming X11 and gaining traction across the industry, but X11 is far from its end of life due to its early and widespread adoption. Even now, X11 is the default graphics protocol for most unix systems, and can easily be installed on any

other relevant OS. This also contributes to the popularity of X11 Forwarding, since a server admin can expect X11 to be configurable on both the client and server machines with little to no additional work. X11 Forwarding is especially popular in computing heavy industries like finance, and is still used for High Performance Computing as it was designed for.

Client-Server model

X11 uses a client-server model, where an X Server is a program on a machine which manages access to graphical displays and input devices (monitors, mice, keyboards, etc.), and an X Client is a program which handles graphical data. With this model, an X Client application can form a connection to an X Server to communicate with the X Server's devices through graphical primitives. This client-server terminology can be confusing in remote scenarios, so remember that in most situations an X Server runs on the user's machine and the X Client runs on the remote machine.



X11 was designed to be network transparent, so that X Servers and X Clients can communicate over local and remote networks in the same way. This can be done by linking an X Server to an exposed tcp address rather than the default localhost or unix socket. However, X11 is an insecure plaintext protocol by default, so it's not recommended to expose an X Server directly. Instead, most users today use X11 Forwarding to take advantage of the security of SSH when running X11 programs remotely.

Display

In X11, a display refers to a group of display devices which an X Server can directly send and receive graphical data. An X Display is generally made up of at least one screen, keyboard, and pointer device. In this context, a screen is not a physical monitor, rather a virtual canvas which can read raw graphical data. In practice a single screen can be made up of multiple monitors and other virtual displays.

X Client Programs use the `$DISPLAY` variable, which looks like `hostname:display_number.screen_number`, to determine which X Display to connect to. An X Program can derive a `tcp` or `unix` socket from this value to form a connection to the display through the X Server. Once the connection is accepted, the X Server forwards the connection to the requested screen.

The `$DISPLAY` variable has some hidden rules that can be a bit confusing. First of all, the display number must always be explicitly set, while the hostname and screen number will default to `device_name/unix` and `0` respectively. As a result, `:0` is actually `device_name/unix:0.0`, and the two values will be treated identically. You can also use `unix:0` to refer to `device_name/unix:0`.

Secondly, a display's associated tcp or unix socket is derived like so:

- `hostname:n` -> `localhost:6000+n`
- `hostname/unix:n` -> `/tmp/.X11-unix/Xn`

Security

There are a few ways that an X Server can control access to its displays, but the most common one, and the only relevant one to X11 forwarding, is cookie-based access with the protocol `MIT-MAGIC-COOKIE-1`. In this protocol, an X Client must provide a valid plaintext 32-byte cookie. If the X Server recognizes the cookie for the requested display, it will provide the client with access to the display with the set of permissions allowed for that cookie. Unfortunately, these permissions are

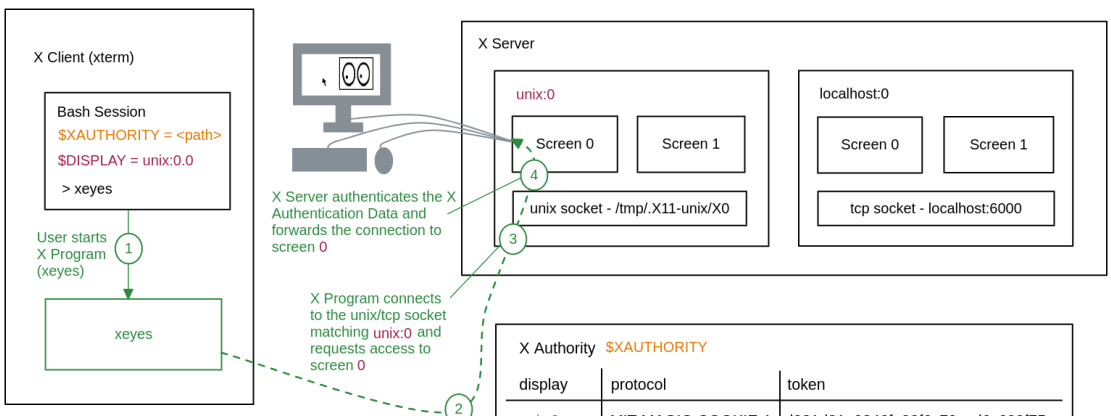
not fine-grained but rather split up into only two categories; trusted and untrusted. A cookie with trusted permissions will provide unmitigated access to the X Server, while an untrusted cookie will restrict permissions, such as restricting the program to only its own window and denying access to the clipboard.

Using the `xauth` program, you can add and generate cookies in an X Server and save them to disk to `$XAUTHORITY` if set, or `~/.Xauthority` otherwise. When you run an X Program, it will retrieve X Auth data for the requested display from `$XAUTHORITY` or `~/.Xauthority`, and provide the X Auth data when connecting to the X Server in order to be authenticated.

One thing you should know is that if an X Program cannot find any X Auth data for the requested display, it will form the connection without X Auth data. The X Server will just accept the connection anyways and uses its default insecure connection method. This means that it is the sole responsibility of the X Program to enforce its own authentication and authorization, rather than the X Server enforcing it. For this reason, `xauth` is usually used alongside other access control systems, such as `xhost`, to prevent untrusted X Clients from even attempting to connect to the X Server.

X Program

Now that each component of the X Window System has been explained, we can take a look at how it all ties together when you run an X Program like `xeyes`.



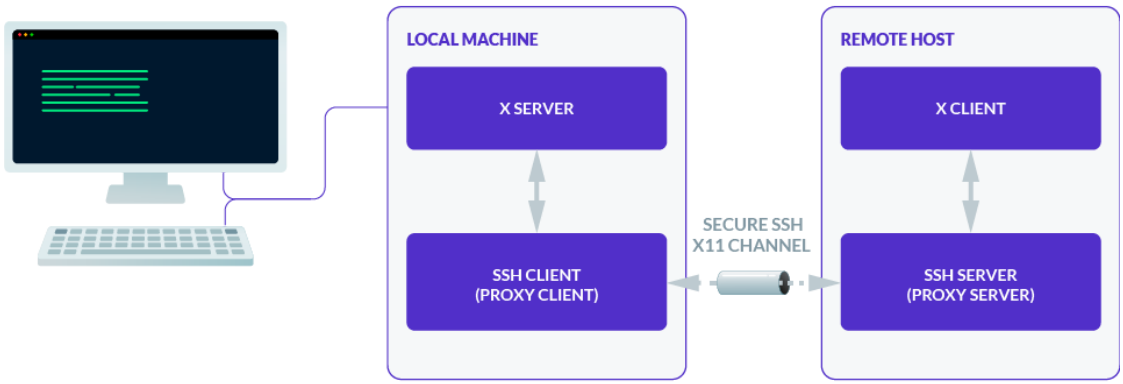
X Program retrieves X Authentication protocol and token for unix:0

unix:0	MIT-MAGIC-COOKIE-1	d021d31e0346fa93f6c70ecd6c609f75
localhost:0	MIT-MAGIC-COOKIE-1	c94f2bcc409e9bb9c36c43941983aa57

How does X11 Forwarding work?

With the basics of X11 out of the way, we can dig into the details of X11 Forwarding.

X11 Forwarding follows the same model as X11, but the X Client to X Server connection gets tunneled through an SSH Channel. In order to achieve this flow, the SSH Server proxies the remote X Client connection to the SSH Client, and the SSH Client proxies it to the user's X Server.



Sounds straightforward enough, but the SSH server and client do a lot of work behind the scenes to make sure an arbitrary X Program in an SSH session gets successfully and securely forwarded to your local X Server. To uncover these secrets of X11 Forwarding, we'll first see how we go from `ssh -X` to a fully set up X11 Forwarding SSH session. Second, we'll see how an X Client Program running in this session manages to communicate with your local X Server. Lastly, I'll explain how X11 Forwarding utilizes the local and remote X Authorities to prevent just anyone on the remote host from accessing your local X Server, as well as how Untrusted X11 Forwarding (`ssh -X`) differs from Trusted X11 Forwarding (`ssh -Y`) to protect your local X Server from malicious remote users and hackers.

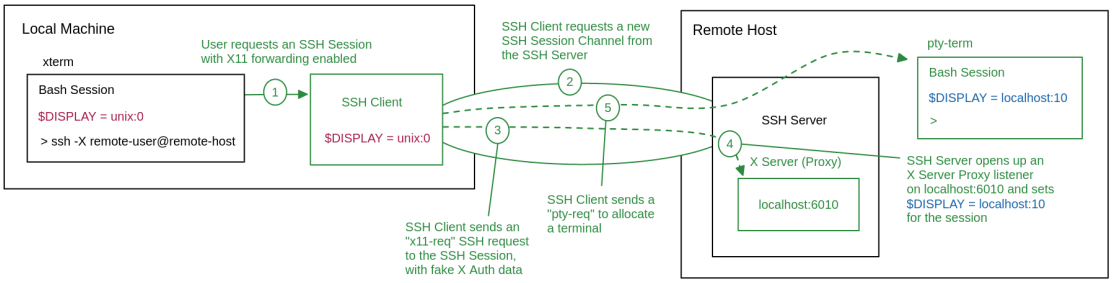
X11 Forwarding session setup

When you enter the command `ssh -X remote-user@remote-host`, your machine will

start up a new SSH Client and request an SSH Session from the remote SSH Server `remote-host`, the same way it would if you omitted the `-X` flag. Once the normal SSH Session is created, your SSH Client will follow it up with an `x11-req` SSH request. If X11 Forwarding is allowed by the SSH Server for the `remote-user`, then it will begin setting up X11 Forwarding for the session.

The SSH server will open up an X Server Proxy listener starting from `localhost:6010` and sets the SSH Session's `$DISPLAY` to the corresponding socket `localhost:10`. As a result, any X Programs started within the session will look at `$DISPLAY` and connect to the X Server Proxy's tcp socket. The SSH server can then intercept the X11 connection and forward it to the SSH client.

Once the server signals to your SSH client that X11 Forwarding was successfully set up, the SSH client will send a `pty-req` SSH request to start a new bash session on the remote host, finalizing the SSH session setup as normal.

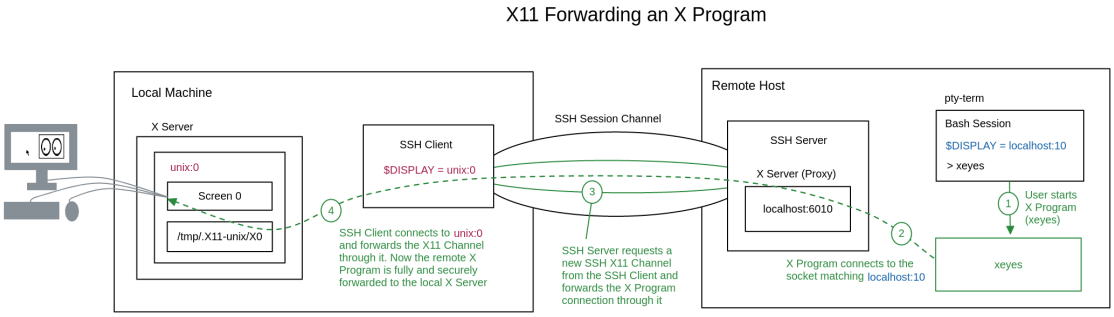


Running an X Program in an X11 Forwarding session

Now that we have a fully set up X11 Forwarding SSH session, we can see what happens when you run an X Program like `xeyes`. Since the remote Bash session has `$DISPLAY=localhost:10` set, `xeyes` will connect to the X Server Proxy on `localhost:6010`. When the SSH server receives this connection, it will create an `x11` SSH Channel for the session and start forwarding the `xeyes` connection into the channel to the SSH client.

The SSH client will then connect to your local `$DISPLAY` and forward the `xeyes`

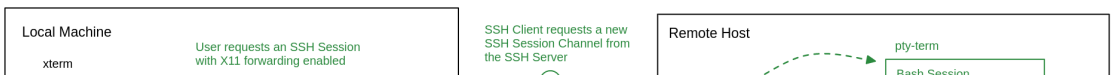
connection from the X11 SSH Channel through this connection. As usual, your X Server will forward this connection to the correct screen and its assigned display devices, thus completing the connection from the remote `xeyes` program to your local display and I/O devices.

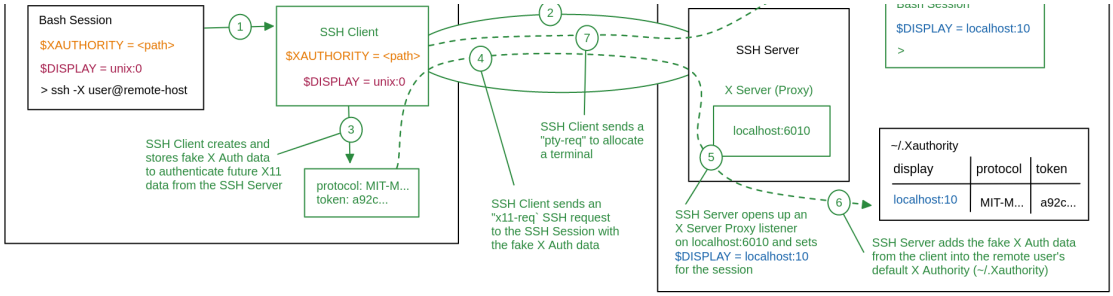


X11 Forwarding X Authority setup

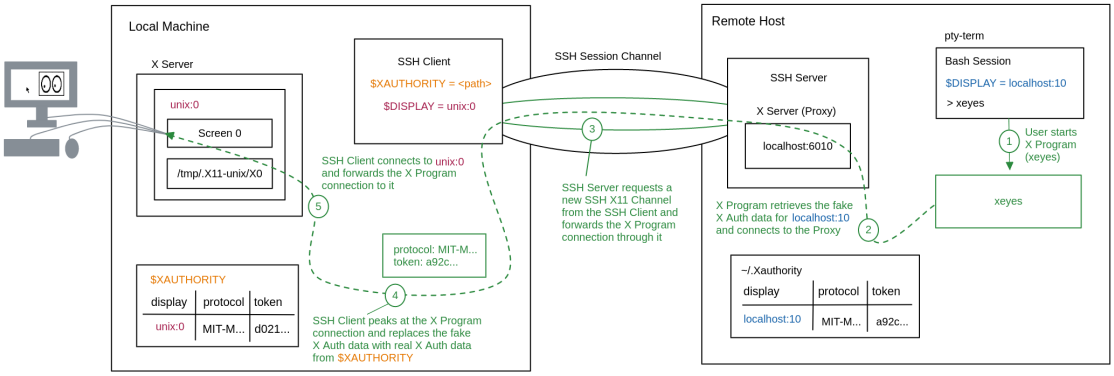
As I alluded to above, the X11 Forwarding described so far is missing a crucial piece of the puzzle. As it is currently, the X11 Forwarding Session will provide anyone that can access the X Server Proxy at `localhost:6010` on the remote host with access to your local X Server. A malicious user on the remote machine could use this to forward an X Program that could take screenshots of your display, capture your mouse and keyboard actions, or even inject X11 actions to run its own commands in an open terminal. To circumvent this, we need to provide the SSH client with a way to authenticate requests that come from the remote X Server Proxy.

To authenticate forwarded X Program connections, the SSH client will attach an X Authority protocol and cookie to the original `x11-req`. The cookie won't actually be used to connect to your X Server, so the SSH Client can just generate a random fake cookie for the `MIT-MAGIC-COOKIE-1` protocol and keep a record of it before sending it in the request. Upon receiving the fake X Auth data, the SSH server will add it to the remote X Authority at `~/.Xauthority` for `$DISPLAY=localhost:10`.





This time when you run the `xeyes` program, it will retrieve the fake X Auth data from `~/.Xauthority` and attach it to the X Server connection. After tunnelling the connection through the X11 Channel, the SSH client will read the X Auth data from the initial X11 data in the connection, and decide whether or not to accept the connection. If accepted, the SSH Client will retrieve real X Auth data from your local `$XAUTHORITY` to replace the fake X Auth data in the connection. Finally, the SSH client will forward the connection to your `$DISPLAY` like before.



Trusted X11 Forwarding `ssh -Y`

The X Authority setup explained so far is called Trusted X11 Forwarding because you are ultimately trusting the access controls of the remote host to protect access to the X Server Proxy. If someone else has access to `remote-user` or `root`, they can access `/users/remote-user/.Xauthority` and use it to run X Programs connected to your local X Server. This would expose your X Server to the X11 attacks mentioned before. In order to protect your X Server from these attacks, you'll need to use Untrusted X11 Forwarding.

Untrusted X11 Forwarding `ssh -X`

In Untrusted x11 Forwarding, we perform one extra step before starting X11 Forwarding. On the local machine, the SSH client creates a new temporary `$XAUTHORITY` and uses `xauth` to create a new untrusted cookie. Now, when the SSH client reads and replaces the fake X Auth data from a forwarded X11 connection, it will replace the fake data with this untrusted cookie instead of using the default trusted settings.

Anyone who connects to `localhost:6010` and shows the correct fake X Auth data will only be granted untrusted access. While this does mean that malicious users can potentially still run an X Program and forward it to your X Server, any potentially damaging X11 permissions will be restricted. It's still far from a perfect system, but overall Untrusted forwarding provides a solid security standard and avoids the more serious risks associated with Trusted forwarding.

Unfortunately, Trusted forwarding is actually the default behaviour of `ssh -X` on unix OS's. There are a variety of reasons for this, but to summarize, the untrusted permissions system was added as a plugin after the fact rather than being built into the X11 protocol itself. As a result, Untrusted forwarding has worse performance as well as compatibility issues, and overall it is less intuitive and straightforward to use. Although the performance and compatibility issues only rarely rear their ugly heads today, 20 years ago when X11 Forwarding was added to OpenSSH, it was a major issue. Due to this unfortunate history, you need to explicitly set `ForwardX11Trusted no` in your SSH client config `~/.ssh/config` to start Untrusted forwarding, which I very strongly recommend. You will still be able to use `ssh -Y` if you absolutely need to use Trusted forwarding for a specific use case.

Conclusion

X11 forwarding is a useful and reliable tool for running graphical programs on remote machines securely. It's also relatively easy to set up, which is important for scaling remote graphical access across wide and varied server fleets. In this post, we looked at some of the lesser known details about X11 forwarding, how it works, and how it can be used securely.

In this blog post, I also mentioned that there are more performant remote graphical protocols, with their biggest weakness being setup and security. Teleport aims to be the most secure and widely accessible remote access plane, so you can expect VNC and RDP integration to come some time in our future. In fact, we just recently started getting requested to implement both. To stay up to date with Teleport's latest developments, keep an eye on our GitHub page.