CS 161 Computer Security
Spring 2010 Paxson/Wagner

Notes 2/19

The Motivation for Firewalls

Suppose you are given a machine, and asked to harden it against external attack. How do you do it?

One starting point is to look at the network services that this machine is providing to the outside world. If any of its network services are buggy or have security holes, an attacker may be able to penetrate your machine by interacting with that application. As we know, bugs are inevitable, and bugs in security-critical applications often lead to security holes. Thus, the more network services your machine runs, the greater the risk.

This suggests one simple way to reduce the risk of external attack: *Turn off every unnecessary network service*. Disable every network-accessible application that isn't absolutely needed. Build a stripped-down box that runs the least amount of code necessary; after all, any code that you don't run, can't harm you. And for any network service that you do have to run, double-check that is has been implemented and configured securely, and take every precaution you can to render its use safe.

This is an intuitive and effective approach, and it can work well when you only have one or two machines to secure, but now let's consider what happens when we scale things up. Suppose you are in charge of security for all of Macrosloth Corp. Your job is to protect the computer systems, networks, and computing infrastructure of the entire company from external attack. How are you going to do it?

If the company has thousands of computers, it won't be easy to harden every single machine individually. There may be many different operating systems and hardware platforms. Different users may have vastly different users, and a service that can be disabled for one user might be necessary to another user's job. Moreover, new machines are bought all the time, machines come and go every day, and users upgrade their machines. At this scale, it is often hard even to get an accurate list of all machines inside the company—and if you miss even one machine, it is then a vulnerable point that can be broken into and might serve as a jumping-off point for attackers to use to attack the rest of your network. The sheer complexity of managing all of this might make it infeasible to harden each machine individually.

Nonetheless, it's still true that one risk factor is the number of network services that are accessible to outsiders. This suggests a defense. If we could block, *in the network*, outsiders from being able to interact with many of the network services running on internal machines, we could reduce the risk. This is exactly the concept behind *firewalls*: the firewall is a device designed to block access to network services running on internal machines.

At this point, it's clear that there are two questions we'll have to settle:

1. What is our *security policy*? For example: Which network services should be made visible to the outside world, and which ones should be blocked? How do we distinguish insiders from outsiders?

2. How will we enforce this security policy? How do we build a firewall that does what we want? What are the implementation issues?

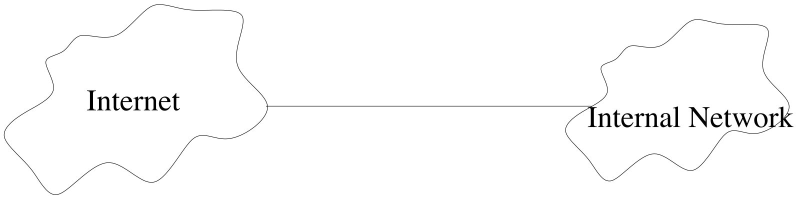

Figure 1: An example network topology, with a single link connecting the internal and external networks.

We'll tackle these each on their own.

1 Security Policy

A little bit of background. In its simplest form, we can visualize the topology of the internal network as shown in Figure 1. We have an internal network, which hosts all the company's machines, and the external world (e.g., the rest of the Internet), and a communications link between the two.

How do we decide what is inside, and what is outside? We might decide that we trust all company employees, but we don't trust anyone else (a very simple threat model). Then we'll define the internal network to contain machines owned by trusted employees, and the external world to include everything else. The link to our Internet Service Provider (ISP) might be the link between these two worlds.

The very simplest security policy is an *outbound-only* policy. Let's distinguish between inbound and outbound connections. *Inbound connections* are initiated by external users and attempt to connect to services running on internal machines, while *outbound connections* are attempts by internal users to initiate contact with external services. An outbound-only policy would permit all outbound connections (reasoning: internal users are trusted; if they want to open a connection, we'll let them), but all inbound connections would be strictly denied. The effect is that none of our network services are visible to the outside world, though of course they can still be accessed by internal users. Unfortunately, this policy is probably too restrictive for any large organization, since it means that the company cannot run a webserver, a mail server, an FTP server, and so on. Therefore, we will need a little more flexibility in how we define the security policy.

In general, the security policy is going to be a particular kind of *access control policy*. We will have two subjects: an anonymous external user, and a generic inside user.[1] The objects are the set of network services that are run on all inside machines; if there are 1,000 machines, and each machine runs 5 network services, we end up with 5,000 objects. The access control policy should then specify, for each subject and each object, whether that subject has permission to access that object.

Firewalls are usually used to enforce a particularly simple kind of access control policy. Inside users are permitted to connect to any network service desired. External users are restricted: there are some services that are intended to be externally visible, and external users are permitted to connect to these services, but there are also other services that are not intended to be accessible from the outside world, and those services are blocked by the access policy.

The first thing the security administrator needs to do is identify a security policy, or in other words, which services external users should and shouldn't be given access to. How should we do it? Broadly speaking, there are two philosophies we might use to determine which services we allow external users to connect to:

[1] Alternatively, we could say that the subjects will be divided into two groups. The inside users group contains all company employees, and the external users group contains everyone else. In our case, the access granted to a subject will be determined solely by which group they are in.

- *Default-allow*: By default, every network service is permitted, unless it has been specifically listed as denied. Under this approach, one might start off by allowing outside users access to all internal services, and then mark a few that are known to be unsafe and should be blocked. For instance, if tomorrow we hear about a new threat the targets Internet Relay Chat (IRC) servers, then we might revise our security policy by denying outsiders access to our IRC servers.
- *Default-deny*: By default, every network service is denied, unless it has been specifically listed as allowed. We might start off with a list of a few known servers that need to be visible to the outside world and that have been adjudged to be reasonably safe; external users will then be implicitly denied access to any service not on the list. If our users complain that, say, their department's FTP server is not accessible to the outside world, we can check whether they are running a reasonably safe and properly configured implementation of the FTP service, and if so add them to the "allow" list.

A default-allow policy is a lot more *convenient*, because from a functionality point of view, everything stays working. However, from a security perspective, default-allow is seriously flawed. The problem is that default-allow fails open: if you make any mistake (i.e., there is some service that is vulnerable, but you forget to add it to the "deny" list), then the result is likely to be an expensive security failure.

In comparison, default-deny fails closed: if you make a mistake (i.e., some service that is safe has been mistakenly omitted from the "allow" list), then the result is merely a loss of functionality or availability.

In addition, when operating at large scales such errors of omission are likely to be common.[2] Because errors of omission are a lot more dangerous in a default-allow policy than in a default-deny policy, and because the cost of a security failure is often a lot more than the cost of a loss of functionality, default-deny is usually a much safer bet.

Default-deny has another advantage. When the system fails open, you may never notice the failure. Attackers who penetrate your system security are unlikely to tell you that they have done so, and so security breaches may go unnoticed for a long time. This gets you into an arms race, where you have to keep up with all the attacks adversaries discover, and even stay ahead of them. This arms race is generally a losing proposition, because there are a lot more of the attackers than there are of the defenders, and the attacker only has to win once to make you really miserable. In contrast, when the system fails closed, someone will probably notice a configuration error (they'll complain: *why isn't the FTP service working?*), and the omission will be immediately evident and easily correctable. This makes failures in default-allow systems that much more costly than failures in default-deny systems.

For these reasons, almost all well-implemented firewalls use a default-deny policy. The security policy specifies a list of "allowed services" which external users are permitted to connect to, and all other services are forbidden. In many cases, some kind of risk assessment and cost-benefit analysis is applied to every network service on the allowed list; if some service is too risky compared its benefits, then it is removed from the allowed list.

How can we identify network services? Let's recall some background on TCP/IP networks. A TCP service is recognized by the machine's IP address and the TCP port number on that machine. For instance, the web server on `www.cs.berkeley.edu` (currently) resides at IP address `128.32.244.172`, port `80`. The mail service resides at `169.229.60.161`,[3] port `25`. UDP services are identified similarly, though of

[2] Indeed, errors of omission are a lot more likely than errors of commission. There may be thousands of potential services out there, but only a few dozen are likely to make your deny or allow list. This means there are thousands of chances to inadvertently omit a service from the list (an error of omission), but only a few dozen chances to inadvertently put something on the list that doesn't belong there (an error of commission).

[3] Actually, there are four mail servers, of which this is one.

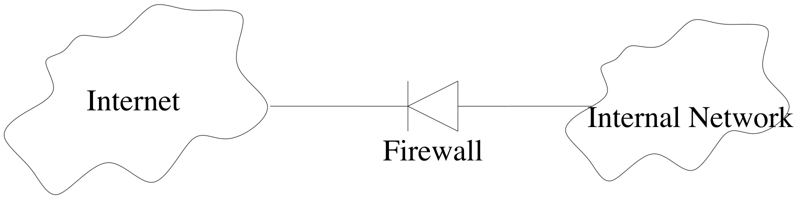

Figure 2: A network topology with a firewall restricting inbound access.

course the port namespace for TCP and UDP is disjoint (a TCP service on port 25 is different from a UDP service on port 25).

Therefore, we can identify each network service with a triplet (m, r, p), where m is the IP address of a machine, r is a protocol identifier (e.g., TCP or UDP), and p is the port number. For instance, the company might have its official web server hosted on machine `1.2.3.4`, and then (`1.2.3.4`,TCP,`80`) would be added to the allowed list. In a default-deny policy, the list of network services that should be externally visible would be represented as a set of these triplets.

2 Enforcement: Packet filters

The key trick behind enforcing such a security policy is to do it at a *chokepoint* in the network. In Figure 1, there is only a single link connecting the inside and outside networks. Therefore, we will replace that link with a firewall that filters network connections and blocks any connections that are denied by the security policy. See Figure 2 for the result.

The existence of a central chokepoint gives us a single place to monitor, where we can easily enforce a security policy on thousands of machines with minimal effort. The idea is a familiar one from physical security. For instance, at the airport, all passengers are funneled through a security checkpoint where access can be controlled. It's a lot easier to perform such checks at one or a few checkpoints than at dozens or hundreds of entrances.

The simplest kind of firewall is a *packet filter*. A packet filter is a router that is augmented with an access control list, usually specified as a list of rules. When any packet is received by the router, the security rules are consulted to decide whether the packet should be forwarded or should be dropped. A rule can specify which packets it will apply to, based on the header fields of the packets. For instance, the rule might specify source and destination IP addresses and port numbers and protocol names, or wild cards for each of these. Each rule also specifies what action to take for matching packets; typical values might be ALLOW or DROP.

As each packet is processed, the list of rules is examined one-by-one, and the *first* matching rule determines how the packet will be handled.

Let's try an example. What does this ruleset do?

```
drop  tcp *:* -> *:23
allow  *  *:* -> *:*
```

Answer: it blocks all TCP packets destined to port 23[4] and forwards all other traffic undisturbed. Notation: `1.2.3.4:25` indicates IP address `1.2.3.4` and port `25`; $\star$ is a *wildcard*, which may appear anywhere.

[4] Port 23 happens to be the Telnet port, use for remote login. Telnet is a major security hole because it transmits passwords in cleartext.

One problem with this policy is that it has no notion of a connection, or of inbound vs outbound connections. It will drop outbound Telnet connections initiated by inside users, which might be undesirable. Another problem is that this is a default-allow policy: it allows everything except one explicitly listed service. As we've argued before, that's error-prone.

So let's suppose that we've carefully built a security policy and decided that we want to allow inbound connections to port 25 of our mail server (`1.2.3.4`), and allow all outbound connections, and that's it. Let's suppose we've predefined a macro `{ourhosts}` as some hard-coded list of all internal hosts. How does this ruleset look?

```
allow tcp *:* -> 1.2.3.4:25
allow tcp {ourhosts}:* -> *:*
drop   *  *:* -> *:*
```

Answer: This policy doesn't do what we want, because of the way that TCP connections work. Recall that a TCP connection is bidirectional, and involves packets going in both directions. Indeed, when a TCP connection is initiated, the initiator sends a SYN packet (i.e., the `SYN` bit in the TCP header is set), the responder responds with a `SYN+ACK` packet (i.e., both the `SYN` and `ACK` bits are set in the TCP header), and the initiator sends an ACK packet; finally, both are able to send data in either direction, and all of the packets other than the very first SYN packet have the ACK bit set.

The problem is now evident: outbound connections aren't actually going to work. If some inside host tries to open a TCP connection to port 80 on an external machine (say), then the initial SYN packet is going to get through, because it is allowed by rule 2. However, the `SYN+ACK` packet coming back does not match rule 1 (because it isn't destined to port 25) and does not match rule 2 (because the source IP address is that of the external web server, not an inside host), so it will be dropped by rule 3, and the connection attempt will time out and fail.

What we *want* is that inbound packets associated with an outbound connection should be allowed, but inbound packets associated with an inbound connection need to be restricted. We need some way to distinguish the two kinds of inbound packets.

The trick is to use a feature of TCP: the very first packet does not have its ACK bit set, but all other packets do. Moreover, the recipient will discard any TCP packet with its ACK bit set, if the packet is not associated with an existing TCP connection. Therefore, the solution is to use a ruleset like this:

```
allow tcp *:* -> 1.2.3.4:25
allow tcp {ourhosts}:* -> *:*
allow tcp *:* -> {ourhosts}:* (if ACK bit set)
drop   *  *:* -> *:*
```

Rules 1 and 2 allow inbound connections to port 25 on machine `1.2.3.4`; rules 2 and 3 allow outbound connections to any port.

Why does this work? Suppose that the attacker discovers a vulnerability in our "Finger" service (TCP port 79), and tries to open an inbound TCP connection to a Finger server on some internal machine. They will have to send a SYN packet to the internal machine, but because this packet does not have its ACK bit set, it will not be allowed by the policy and will be dropped before it is ever seen by the internal Finger server program. If they try to send a packet with its ACK bit set (say, a `SYN+ACK` packet) to port 79 on the internal machine, this will be permitted by the firewall, but all TCP stacks disregard any arriving packet that has its ACK bit set, but is not part of an existing connection, so such packets will also be harmless. In this way, we can specify policies restricting inbound connections arbitrarily.

However, there is a subtle security hole lurking in this ruleset. The problem is related to *IP spoofing*. Recall that there is nothing in the IP protocol that prevents an attacker from sending a packet with an incorrect (*spoofed*) source address; indeed, routers in general don't even look at the source address, so such a packet will be correctly routed to the destination. Suppose that `1.2.3.7` is one of our internal hosts. Imagine if an attacker sends a spoofed TCP SYN packet, with its source address set to `1.2.3.7`, the destination address of some targetted internal machine, and destination port 79. Note that this packet will be allowed by rule 2 of the ruleset above, and so will reach the internal target machine. The target host will respond with a SYN|ACK packet to `1.2.3.7` and wait for the ACK that completes the three-way handshake. If the attacker at this point sends a spoofed TCP ACK packet, with source address `1.2.3.7`, and then follows it up with a spoofed TCP data packet, again with source address `1.2.3.7`, then all of these packets will be allowed through the packet filter and will be accepted by the target host (because they are part of a valid TCP connection).[5] This allows the attacker to connect to internal hosts, in violation of our security policy, and would allow the attacker to exploit any security holes in the Finger service of which they are aware. That's a serious problem.

The fix is for the packet filter to mark each packet with the interface it arrived on, and to allow rules to match on this interface. Remember that a router is a device that has two (or more) interfaces, where network *links* can be plugged in; a packet is received on one interface, and is forwarded out on another interface. Suppose that we call the interface attached to the internal network `in`, and the interface to the rest of the Internet `out`. Then we can revise our ruleset as follows:

```
allow tcp *:*/out -> 1.2.3.4:25/in
allow tcp *:*/in  -> *:*/out
allow tcp *:*/out -> *:*/in    (if ACK bit set)
drop   *  *:*     -> *:*
```

This ruleset allows inbound packets only if they are destined to host `1.2.3.4`, port 25 (rule 1) or if they have their ACK bit set (rule 3); all other inbound packets are dropped. This is a clean solution that defeats the IP spoofing threat. It also happens to simplify ruleset administration, as we no longer need to hardcode the list of IP addresses of internal machines.

3 Other Kinds of Firewalls

Packet filters are the crudest kind of firewall: they operate at the network level, and generally look only at TCP, UDP, and IP headers. One can also build firewalls that restrict traffic according to the contents of the data fields; these are known as *application-layer firewalls*, or application firewalls for short. Application firewalls have some security advantages, because they can enforce more restrictive security policies and because they can transform data on the fly.

For more information on firewalls, the authoritative reference is Cheswick, Bellovin, and Rubin: *Firewalls and Internet Security: Repelling the Wily Hacker*. Packet filtering software is available for many operating systems: e.g., Linux has `iptables`, OpenBSD/FreeBSD has `PF`, and Windows XP and Vista have their own firewalls.

[5] Here we are assuming that the attacker can correctly guess the Initial Sequence Number (ISN) that the server chooses for this connection. As discussed previously in lecture, originally attackers could do so because the ISN generation algorithm was based on the server consulting its local clock. Today, such numbers are generated in an effectively random fashion, making this attack much harder. However, the discussion here remains relevant when considering firewall rules for UDP traffic, since UDP traffic lacks such sequence numbers and thus is much easier for an attacker to correctly spoof.

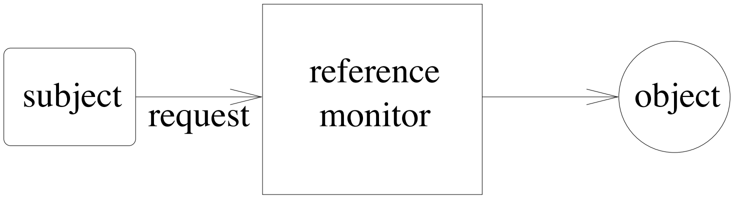

Figure 3: A reference monitor controls access by subjects to objects.

4 Principles

Firewalls embody several useful principles that you can apply elsewhere in computer security. As mentioned earlier, firewalls can be thought of as enforcing a particular kind of access control policy, and they are optimized for this special task. The notion of a chokepoint is crucial, because it is what makes it possible to enforce the access control policy, and we see the same thing come up elsewhere in access control.

In general, the mechanism that enforces an access control policy often takes the form of a *reference monitor*. The purpose of a reference monitor is to examine every request to access any controlled resource (an "object") and determine whether that request should be allowed. See Figure 3.

There are three security properties that any reference monitor should have:

- *Always invoked:* The reference monitor should be invoked on every operation that is controlled by the access control policy. There must be no way to bypass the reference monitor. This is sometimes also known as the *complete mediation* property: all security-relevant operations must be mediated by the reference monitor.
- *Tamper-resistant:* The reference monitor should be protected from tampering by other agents. For instance, other parties should not be able to modify its code or state. The integrity of the reference monitor must be maintained.
- *Verifiable:* It should be possible to verify the correctness of the reference monitor, including that it actually does enforce the desired access control policy correctly. This usually requires that the reference monitor be extremely simple, as generally it is beyond the state of the art to verify the correctness of subsystems with any appreciable degree of complexity.

We can recognize a firewall as an instance of a reference monitor. How are these three properties achieved?

- *Always invoked:* We assumed that the packet filter is placed on a chokepoint link, with the property that all communications between the internal and external networks must traverse this link. Thus, the packet filter has an opportunity to inspect all such packets. Moreover, packets are not forwarded across this link unless the packet filter inspects them and forwards them (there needs to be no other mechanism by which packets might flow across this link).

  Of course, in some cases we discover that it doesn't work out we hoped. For instance, maybe a user hooks up an unsecured wireless access point to their internal machine. Then anyone who drives by with a wireless-enabled laptop effectively gains access to the internal network, bypassing the packet filter. This illustrates that, to use a firewall safely, we'd better be sure that our firewalls covere *all* of the links between the internal network and the external world. This set of links is termed the *security perimeter*.

- *Tamper-resistant:* We haven't really discussed how to make packet filters resistant to attack. However, they obviously should be hardened as much as possible, because they are a single point of failure. Fortunately, their desired functionality is relatively simple, so we should have a reasonable chance at protecting them from outside attack. For instance, they might not need to run a standard operating system, any user-level programs, or network services, eliminating many avenues of outside attack. More generally, we can use firewall protection for the firewall itself, and not allow any management access to the firewall device except from specific trusted machines. Of course, the physical security of the packet filter device must also be protected.

- *Verifiable:* In current practice, unfortunately the correctness of a firewall's operation is generally not verified in any systematic fashion. The software is usually too complex for this to be feasible. And we do suffer as a result of our failure to verify packet filters: over time, there have been bugs that allowed attackers to defeat the intended security policy by sending unexpected packets that the packet filter doesn't handle quite the way it should.

The notion of a reference monitor recurs over and over again. Thus, the three requirements for a secure reference monitor are well worth absorbing.

Firewalls also embody another useful principle: *Orthogonal security.* If the security mechanism is orthogonal from, and transparent to, the rest of the application, then it can be deployed to protect pre-existing legacy systems much more easily than security mechanisms that must be integrated with the rest of the system. A reference monitor that filters the set of requests, dropping unallowed requests but allowing allowed requests to pass through unchanged, is essentially transparent to the rest of the system: other components do not need to be aware of the presence of the reference monitor. Such mechanisms are easier to retrofit into legacy systems.

However, while orthogonal security has nice deployment properties, it also represents a form of "bolt-on security": that is, security that's added to a system after the system has already been designed and implemented. Bolt-on security can prove brittle because if the design of the added security mechanism uses different abstractions than those of the system it's meant to protect, there can be holes in the protection. We will discuss some of these in lecture in terms of ways that firewalls can be evaded due to their limited understanding of the traffic they carry.

5 Experience with Firewalls

Firewalls have been tremendously widely used. They are a great success story of technology transfer from research to practice. The first firewall paper was published at a research conference in 1990, and within a few years firewalls were widely used.[6]

Why do firewalls work well?

- *Central control:* A firewall provides a single point of control. When security policies change, only the firewall has to be updated; individual machines do not need to be touched. For instance, when a new threat to an Internet service is discovered, it is often possible to very quickly block it by modifying the firewall's security policy slightly, and all internal machines benefit from this protection. This makes it easier to administer, control, and update security policy for an entire organization.

[6] For example, the company Checkpoint was founded in 1993. Today it has the greatest market share in firewalls and brings in nearly $1 billion per year in revenue.

- *Easy to deploy:* Because firewalls are essentially transparent to internal hosts, there is an easy migration path, and they are easy to deploy (incrementally, or all at once). Because one firewall can protect thousands of machines, they provide a huge amount of leverage.

- *An important problem:* They address a burning problem. Security vulnerabilities in network services are rampant. In principle, a better response might be to clean up the quality of the code in our network services; but that is an enormous challenge, and firewalls are much easier.

Firewalls have some serious shortcomings, though. How do firewalls fail? What are their disadvantages?

- *Loss of functionality:* The very essence of the firewalls concept involves turning off functionality, and often users miss the disabled functionality. Some applications don't work with firewalls. For instance, peer-to-peer networks have big problems: if both users are behind a firewall, then when one user tries to connect to another user, the second user's firewall will see this as an inbound connection and will usually block it.

  The observation underlying firewalls is that connectivity begets risk, and firewalls are all about managing risk by reducing connectivity from the outside world to internal machines. It should be no surprise that reducing network connectivity can reduce the usefulness of the network.

- *The malicious insider problem:* Firewalls make the assumption that insiders are trusted. This gives internal users the power to violate your security policy. Firewalls are usually used to establish a *security perimeter* between the inside and outside world. However, if a malicious party breaches that security perimeter in any way, or otherwise gains control of an inside machine, then the malicious party becomes trusted and can wreak havoc, because inside machines have unlimited power to attack other inside machines. For this reason, Bill Cheswick called firewalled networks a "crunchy outer coating, with a soft, chewy center." There is nothing that the firewall can do once a bad guy gets inside the security perimeter.

  We see this in practice. For example, laptops have become a serious problem. People take their laptop on a trip with them, connect to the Internet from their hotel room (without any firewall), get infected with malware, then bring their laptop home and connect it to their company's internal network, and the malware proceeds to infect all other internal machines.

- *Adversarial applications:* The previous two properties can combine in a particularly problematic way. Suppose that an application developer realizes their protocol is going to be blocked by their users' firewalls. What do you think they are going to do? Often, what happens is that the application *tunnels* its traffic over HTTP (web, port 80) or SMTP (email, port 25). Many firewalls allow port 80 traffic, because the web is the "killer app" of the Internet, but now the firewall cannot distinguish between this application's traffic and real web traffic.

  The fact that insiders are trusted has as a consequence that all applications that insiders execute will be trusted, too, and when such applications are acting in a way that subverts the security policy, the effectiveness of the firewall can be limited (even though the application developers probably do not think of themselves as malicious). The end result is that, over time, more and more traffic goes over port 25 and (especially) port 80, with firewalls gaining less and less visibility into the traffic that traverses them. As a result firewalls, are becoming increasingly less effective.