Problem Set 5 Solutions

Problem 5-1. Unconditional Commitment (Raquel Hill, Adon Hwang, David Molnar)

A naive commitment scheme that involves a trusted third party would be simple:

- SETUP

  Alice generates OTP and submits it to Ted

- COMMIT

  Alice commits to message x and transmits $\{x \oplus \text{OTP}\}$ to Bob

- REVEAL

  Alice transmits $\{x, \text{OTP}\}$ to Bob
  Ted transmits $\{\text{OTP}\}$ to Bob

Now, clearly, Alice cannot cheat- she must reveal to Bob both the secret x and the OTP used to commit it. This is not quite the polished scheme we desire. Ted must get involved after the setup phase.

Let's do a little better. We describe an unconditional commitment scheme using one-time pads and then show its correctness. We combine the one-time pad scheme and message authentication code presented in class.

- SETUP

  Ted generates one-time pad p, key $K = (a, b)$
  Ted computes $f(p, K) = y = ap + b$
  Ted securely and privately transmits $\{p, b\}$ to Alice
  Ted securely and privately transmits $\{y, a\}$ to Bob

- COMMIT

  Alice commits to message x by securely and privately transmiting $\{p \oplus x\}$ to Bob

- REVEAL

  Alice securely and privately transmits $\{p, b\}$ to Bob

Note that Ted only needs to get involved at setup. We rely on him for consistent delivery of $\{p, b\}$ to Alice and $\{y, a\}$ to Bob. Alice does not need to reveal x to Bob since revealing p is good enough given Bob's knowledge of $p \oplus x$, granted that she does not alter p from her original commitment.

Just after the commit phase, Alice knows $\{x, p, b, p \oplus b\}$ and nothing more. Bob knows $\{y, a, p \oplus b\}$ and nothing more.

Bob cannot guess p or x. To guess either, Bob needs to know either p or x. Given y or a, Bob cannot deduce p: b is also unknown. This secrecy is provided by the one-time pad scheme.

Alice cannot cheat by revealing an alternate p', b' such that y is preserved. She does not know a or y and cannot change p and b in such a way to preserve $y = ap + b$.

The crux of the secure commitment is the fact that one point does not determine a line, from the one-time MAC scheme.

Note that if Ted makes the mistake and gives a to Alice and b to Bob, all bets are off. Alice can now cheat by preserving $y = ap + b$ since she knows both a, b and p.

Problem 5-2. Finding t-bags

4-bag (Daniel Barkalow, Laura Dean, Nickolai Zeldovich

We decided to find a large 4-bag, though not as large as we'd like. k is 510 bits, so p_1=120k+1 is a 517-bit number. (p_2=60k+1, p_3=40k+1, and p_4=30k+1.)

(k = 34323988300653048574909503995406966086347176500716527046
9723172959277159169882802606127982033072727748864815569574042
p018560993999858321906287014145568643855):

p_1 = 411887859607836582898914047944883593036166118008598324563
66870755113259100385936312735357843968727329863777868348885148
2227319279982998628754441697468237262601,

p_2 = 205943929803918291449457023972441796518083059004299162281
833390377556629550192968156367678921984363664931888934174442574
1113659639991499314377220848734118631301,

p_3 = 137295953202612194299638015981627864345388706002866108187
889269183710863667953121042451192813229091099545926222782961716
0742439759994332876251480565822745754201,

p_4 = 102971964901959145724728511986220898259041529502149581140
91695188778314775096484078183839460992181832465944467087221287
0556829819995749657188610424367059315651.

10-bag (Grant Emery, Brian Perrin, Emil Sit)

We found some big 4-bags:

```
 4-bag starting at 100000000000000000752555284595204841511062354354417972681
 4-bag starting at 1000000000000000007525552845952048415110623543545407996041
 4-bag starting at 10000000000000000075255528459520484151106235435454161349801
 4-bag starting at 100000000000000000752555284595204841511062354354541832948241
```

The latter is on the order of 10^{59}. We also found some long ones:

```
10-bag starting at 3944152566113761
 9-bag starting at 25078354201844287344001
```

The basic code used to generate these t-bags is appended. We used the GNU Multi-precision library to do our large number processing. The basic algorithm is to find a prime, and see how big t can be while still satisfying the t-bag definition.

t-bag spacing

We had one major optimization to help us find t-bags quickly. Because of the properties of t-bags, we were able to impose restrictions on candidate starting primes for t-bags. This allowed us to skip over large spaces of numbers that could not possibly be t-bags.

The basic idea is that we have certain divisibility requirements. That is, in a t-bag starting at p_1, we must have:

$$\begin{aligned} 2 &\mid p_1 + 1 \\ 3 &\mid p_1 + 2 \\ &\vdots \\ t &\mid p_1 + t - 1 \end{aligned}$$

In order to satisfy this divisibility requirement, this means that we must have $p_1 \equiv 1 \text{mod lcm}(1, 2, 3, \ldots, \text{t})$. To see this, consider the case where $t = 4$. Then this will give us that:

$$\begin{aligned} p_1 &= 12x + 1 \\ p_2 &= 6x + 1 \\ p_3 &= 4x + 1 \\ p_4 &= 3x + 1 \end{aligned}$$

Right away, this tells us that we need only consider one out of every 12 numbers! However, we can further refine this result. In order for the results of dividing $p_1 + k$ by $k - 1$ to be prime, the results must be odd. We note that if x is equivalent to $1 \bmod 2$, then p_4 will be divisible by 2, so we can assume that x is equivalent to $0 \bmod 2$. Similarly, if x is not equivalent to $0 \bmod 5$, then one of the quotients will be divisible by 5, and thus not prime. Thus x must be a multiple of $2 \cdot 5 = 10$. Combining this with our prior result gives:

$$\begin{aligned} p_1 &= 120x + 1 \\ p_2 &= 60x + 1 \\ p_3 &= 40x + 1 \\ p_4 &= 30x + 1 \end{aligned}$$

In other words, our first prime, p_1 must be congruent to $1 \bmod 120$. Similarly, when we checked for t-bags with larger values of t, we were able to put greater bounds on our modulus. To automate the process, we wrote another program (attached) to find the appropriate modulus for any t.

We then had several basic search strategies:

1. Let $t = 4$. Find the appropriate modulus for generating t-bags. Find the first t-bag. Increment t by 1, repeat. Basically, just find the first t-bag for each t.
2. Let $t = 16$ (or some other large and interesting value of t). Find the appropriate modulus for generating 16-bags. Find the first 16-bag.
3. Start at some ridiculous power of 10, say 10^{23}. Find the first 4-bag. Scale that result by 10, repeat.

While we don't have any concrete results on the distribution of t-bags, it seems plausible that there are an infinite number of t-bags for at least small values of t. This guess is based on a theorem by Dirichlet which states that any sequence $an + b$ has an infinite number of primes in it. The proof involves some very complex number theory. Based on our spacing results, we can see that we are examining arithmetic sequences in our search. So, if these sequences overlap, they will probably do so periodically, giving us an infinite number of t-bags. It is even perhaps possible that they exist for arbitrary t. The problem is that the distribution of primes may not "fit" exactly where we want them to.

Problem 5-3. Inetd DoS Vulnerabilities (Daniel Barkalow, Laura Dean, Nickolai Zeldovich)

The standard UNIX "internet super-server" *inetd* tries to prevent looping services (or services which try to spawn server processes too quickly) from making the rest of the system unusable. This is certainly a valid concern, as a large server process could take up considerable amounts of resources (such as memory, CPU, file descriptors, etc) per copy running at any time for each connection open to that service. The scheme in original implementations of *inetd* counts the number of connections per minute, and if it ever exceeds 40 connections in any one-minute time period, the service is declared to be "looping" and the TCP or UDP socket is closed for 5 minutes, after which time it will be re-enabled subject to the same constraint.

This scheme has an obvious weakness: one can quickly open 40 connections to a given service under inetd on any open system and make inetd terminate the service for 5 minutes. In other words, the host is left open to a denial-of-service attack.

Other attacks exist, such as SYN flooding, but we can defend against them in a number of ways, including SYN cookies (setting the SYN+ACK response sequence number to a cryptographic hash of connection parameters) and random drop for new incoming connections. So, for the rest of this problem we shall only consider full-connection flooding.

Protecting inetd

More modern implementations of *inetd* (e.g, that in FreeBSD) have other parameters which one can set to limit the effect of the full-connection attack. These include limiting the number of connections from a single IP, and limiting the number of server processes that can be running at any given time. The first defends against an attacker who has control of only a single (or a couple of) host(s). Since he has to complete the three-way TCP handshake, he needs to be able to receive responses sent to him. The easiest way to do this is for the attacker to connect from his own address. Our scheme will protect against such attacks by disallowing further connections from a single IP address past the limit. Of course a more resourceful attacker can circumvent this, if he can sniff some network segment. Then he could spoof TCP connections from any IP address in the subnet which he can sniff, and likely defeat our per-IP connection limit. One possible trade-off is the case where both the attacker and the user come from the same IP address (e.g., a proxy server). It might be possible to special-case certain IP addresses or to catch the attacker at the proxy server.

Limiting the maximum number of server processes that are running at any given time, but leaving the connection rate unlimited (or much higher than the standard limit), might solve the problem of *inetd* terminating the service. However, if a new server process being is being spawned for every connection (*nowait* option in inetd.conf) and the server process is rather large, this might give the attacker an easy way to keep starting a fixed number of large programs on the victim host.

As we see, the maximum connection per IP address parameter provides some assurance that an attacker working from just a couple of systems will not be able to mount a successful DoS attack. The maximum number of server processes limit is useful in some situations, where the server process is relatively lightweight, or handles multiple connections without being restarted (e.g., UDP services). A combination of these two limits might provide a practical solution, by replacing inetd with one which supports these limits.

Standalone Daemons

Protecting standalone daemons is a bit more tricky. We would need to either insert our own process between the actual service daemon and the socket, or somehow modify the *accept()* call to only pass through "approved" connections.

The first implementation would simply require our process to listen on a socket, accept connections like inetd with protection features described above, and, if the connection is to be allowed, open another TCP connection on its behalf to the actual server process (perhaps running on a different port, or only bound on the local interface (e.g. 127.0.0.1). This may create problems, since the standalone daemons expect the remote connection endpoint to be the actual host from which the user is connecting. Applications which verify the remote IP address (such as Kerberos, in some cases) might fail for that reason. Logging of connections would contain false information, as every connection would appear to be coming from "localhost". Logging could be fixed by having our intermediate process log the connection appropriately, and disable logging of now-invalid information from the standalone daemon.

The other solution, altering the *accept()* call, would help us avoid some of the problems we encountered above. It might even be possible to "protect" programs which we do not have source for, if they are dynamically linked and we can use LD_PRELOAD to inject our version of *accept()* before that of libc (or libsocket). Essentially, our *accept()* function would run the underlying *accept()* call on the socket (perhaps by using *dlopen()* and associated functions to extract it from libc/libsocket), and then apply the same criteria we described above for a replacement *inetd* server. If the connection passes the criteria, it is allowed and our *accept()* returns the new socket. If it does not pass, we close the connection and block in the underlying *accept()* once again.