WAN-hacking with *AutoHack*
– Auditing security *behind* the firewall

To appear in the 5th USENIX Unix Security Symposium, 6th June 1995

Alec Muffett
Network Security Group
Sun Microsystems
United Kingdom[*]

June 6, 1995

Abstract

This paper is a review of an ongoing project to simplify security auditing of the world-wide TCP/IP network of some thirty thousand hosts, internal to Sun Microsystems.

The paper also examines the issues which this project raises; it details the conception, design, development of, and one year's results gathered from, *AutoHack*, a tool specially created to probe, audit, and produce security reports for, a TCP/IP network of this size.

Introduction.

One of the many problems to beset systems administrators who seek secure their machines is a form of *entropy*. Over periods of time ranging from minutes to months, the *effective security* of a machine attached to a network will diminish.

Even if a host has been "locked down" in accordance with some comprehensive security policy, as time progresses more people will become aware of the host's existence, and hitherto undiscovered flaws in its hardware, software, or inadequacies in the standard to which it was secured, will come to light.

In short: even though the machine *per se* does not change, its defenses weaken as more becomes known about them.

Because information regarding security holes is often slow to propagate amongst less-motivated systems administrators, it is common to find pockets of entropy like this, where otherwise notoriously insecure software is still being used on live (and perversely, often mission-critical) systems, because the software is known to be "stable" (ie: the software is *old*) and its failings are not known to local systems administrations staff.

Further, if the administrators are regularly forced to alter a host's setup in order to serve the needs of a changing user base, issues of misconfiguration invariably arise:

- "dead" accounts which exist in obscure parts of the file-system.
- long-forgotten systems software packages.
- "quick hacks" to system security to fix some emergency situation, never set right...

...and when compounded by problems of management (eg: a knowlegable or aggressive user base, remote administration of widely distributed sites, the generally poor scalability of systems administration tasks) then these basic problems of network security can appear beyond the administrator's control.

The modern response to this situation is to *firewall* your systems, at some level restricting access to your machines on the basis of who is trying to access them, and from where. Although this relieves much of the stress placed upon administration staff, the "unthinkable" question remains:

What if someone breaks through the firewall?

The trouble with firewalls...

Firewalling[CB94, Ran93] your system from the Internet (or otherwise partitioning your network by setting up internal firewalls) brings many problems, possibly the worst of which is that the presence of a firewall encourages *slackness* on the "secure" side of the network.

[*]e-mail: *alec.muffett@uk.sun.com* or *alec@hicom.org*

The misconception held by network users and management alike is that threats to security are "contained" by the firewall's presence, that someone else is "dealing" with security, and that therefore the importance of maintaining security on interior networks is somehow lessened.

It is because of this belief that many (often corporate) networks fit the "crunchy shell with a soft centre" model of network security. Around the core of critical datacentres exists a light, fluffy network full of holes, which in turn is supposedly coated with a rock-hard shell of security – the firewall.

This model is rapidly losing ground as a viable large-scale network architecture in the commercial world. With the burgeoning of the Internet, work practices are changing:

- People would like to be able to work from home, accessing their "desktop" machines via the Internet.
- People want to work nomadically, carrying their work environment (their laptop) with them, reading and writing e-mail from the hotel in which they are staying.
- Companies want to share financial data with each other, their banks, their remotest offices, the people who deliver their goods.
- Large companies "take over" smaller ones and must then subsume an entire network of dubious trustworthiness into their own.

Implementation of the "improvements" mentioned above will require holes to be drilled through the protective shell of your firewall, with two possible outcomes: something gets in through the holes and eats your systems, or all your data leaks out the bottom.

The technology necessary to safely share data with remote parties, or to shepherd third-party traffic from the Internet along your own networks into "semi-public" datacentres, consists largely of bleeding-edge proprietary vapourware, or is sorely behind the times, or is tied up in patent lawsuits.

Herein lies both a problem and an opportunity.

In this period of rapid growth of the Internet's importance and usage, whilst we are lacking the software, technology and standards required to create flexible, "open", and yet *secure* data-sharing network architectures, the tasks that we must undertake to maintain security in the meantime seem obvious:

- Restrict your data sharing.
- Watch your firewall, very, very carefully.
- Harden your network throughout, securing all internal and external interfaces.
- Perform regular auditing to detect new hosts that have been attached to your network.
- Fix the holes that you find.

...several points, all of which revolve around one key requirement: that you are able to comprehensively *audit* the security of all hosts that are connected to the network, and present comprehensible reports of your findings to those with the power to fix any faults that exist.

Given the general desirability of maintaining high levels of system integrity[1], as well as the long-term benefits that could be reaped from early introduction of rigorous, network-wide auditing, it seemed to us[2] that it would be useful to obtain or create a tool that would allow us to audit our internal network from a central location, producing detailed reports that could be fed back to grass-roots systems administration personnel, to help them perform the necessary fixes.

Tools at our disposal.

Security tools can typically be divided into two categories:

proactive – tools which are "defensive" in nature, which are not easily utilised for nefarious purposes.

reactive – analytic "offensive" tools which may be of use to both the systems administrator, and to members of the hacker community.

The first group includes software such as *Tripwire*, password-file "shadowing" systems and so-called "fascist" *passwd* programs (such as *passwd+* or *npasswd*), and software like *S/Key*; programs which strengthen authentication mechanisms or detect anomalous behaviour on your system.

However, to the would-be auditor, the second category is much more interesting, including tools that can effect a break-in to (or similarly compromise) a system, regardless of whether the perpetrator is legally permitted to do so.

Programs of this type (eg: *COPS*, *Crack*, *TAMU*, *ISS*, *ypx*, *nfsbug*, and now *Satan*) are not uncommon, because they are precisely the sort of program written by the hackers who want to break into your system.

[1] ...especially since some sources indicate that 60..80% of computer security incidents are caused by *internal* users.

[2] Sun's Network Security Group

Since the only way to prove the robustness of a system is to attack it, it seemed logical that we needed to find or create a network auditing tool that could probe each of our hosts in turn, in the manner of a hacker, if we were going to harden our intra-network security.

The host-oriented reactive tools described above did not appear entirely suitable for our needs, because most of them are meant to be run *upon the host in question* to check for configurational errors, rather than to attack the host over the network.

The two tools which appeared nearest to what we wanted were:

ISS the "Internet Security Scanner" package by Chris Klaus.

Satan a "Security Analysis Tool for Auditing Networks", by Dan Farmer and Wietse Venema.

The first two releases of *ISS* to USENET were as freeware, consisting of single programs which could serially probe a range of IP-addresses in a variety of ways, producing reports "on the fly".

Aside from the interest it generated by its being the first generally available network probe, *ISS* was also notable because it called upon external programs to provide extra functionality (eg: *ypx*, a NIS passwd-map snarfing tool). However, not long after the initial release, *ISS* became a commercial product, and we chose not to pursue investigation of it any further.

Satan, as described by Dan Farmer and Wietse Venema[FV92] sounded immensely suited to our needs. A configurable network probing tool, capable of digesting information from separate attack modules, generating reports as it goes.

The first problem with *Satan*, however, was that the software was not generally available at the time when this project began[3].

It also appeared[Far94] that the upcoming software wasn't *exactly* suited to our needs. Rather than large-scale auditing and report generation, *Satan* appears to be aimed at investigation of the "web of trust" in network security, building a dependency graph that lists which machines trust which other machines, to some finite tree depth.

Reasoning that all hosts on our network are of equal importance to us[4], we decided that "if you can't break into any of them, then it doesn't matter who trusts whom". Bulk auditing was of greater interest, and the matter of securing "hot-spots" in the network could come later.

[3] May 1994

[4] ...although some are more equal than others...

With this in mind, the goal of our project became clear: create a scalable, extensible tool capable of *wide* (as opposed to *deep*) auditing of the entire network, and with the ability to retrospectively produce informative security reports for any arbitrary list of hosts.

Evil Designs...

AutoHack is a tool which wasn't so much developed as *congealed* from good and bad ideas, and it was constantly re-written until (mostly) only the good ideas remained. Certain design requirements were enforced by the limited resources at our disposal, but it cannot be said that there was ever a preconceived "design plan".

In retrospect this was a good thing; armed with nothing more than the spare cycles on a personal workstation[5], half a gigabyte of free disk space, and the political authority to spot-audit the internal network – the lack of deadlines and initial managerial involvement provided the freedom to experiment, throw away code, and to generally keep going until the software just "looked right".

The goals seemed obvious:

- Simplicity should be pervasive: simple data formats, sensible ordering of information, and easy access to that information.
- The user should be able to create new probes and add them to the suite without having to modify any part of the package, other than the report writer.
- Data received from individual probes should be stored verbatim, to permit incremental improvements to modules (eg: the report writing software) without forcing a complete re-scan of the network.

  (It should be noted that this immediately suggests the separation of function into data-gathering and data-interpreting modules).
- Modularity is very desirable. Apart from the benefits of being able to rewrite or add new modules to the suite without affecting other code, adopting the "processes, pipes and filters" model would reap benefits by allowing us to utilise the existing software base, eg: *tftp* and *rsh* clients, saving time otherwise spent rewriting protocol drivers.

[5] A 32Mb SPARCStation 10/40.

```
#!/bin/sh
while read host
do
        for user in root daemon bin sys smtp adm nobody
        do
                su $user -c "rsh -n $host ´echo $host-$user´"
        done
done
```

Figure 1: Anatomy of *AutoHack* v0.1.

- The user interface should be both comprehensible and idiot-proof, and should provide no additional functionality other than to provide structure to the functionality of the underlying modules, permitting us to throw away or rewrite the user-interface without cost.
- All code should be written with scalability in mind; the worst-case scenario for a network probe is that it should be set to scan the entire Internet; it should either be able to cope with this load with little or (preferably) no modification, or return a sensible error message to the user.
- The data-gathering component should be written so that it can be restricted to using no more resources than are comfortably available, in terms of network bandwidth, etc. Speed is important, but it is not as important as keeping your local network manager happy[6]. If the program can be designed in such a way that complications such as file-locking, etc, never become problems in normal operation, so much the better.

Much of the above appears peripheral to the matter of probing hosts and discovering their security holes, but as we shall discuss, much of the power of *AutoHack* lies in its ability to cope with almost any size of task that is given to it.

The Attack Engine.

As shown in Figure 1, *AutoHack* began as a trivial shell-script probing for the local network for systems with "promiscuous" trust (brought on by NIS wildcards in `/etc/hosts.equiv`, `/.rhosts`, or similar). Once it became apparent that the result from a wider "audit" would be interesting, the program began to evolve.

Problems that needed to be addressed immediately at the start of the project, included:

- The matter of host availability. The connection phase of the *rsh* process could "hang" the script for several minutes at a time if the remote host was unreachable for some reason.

  The simplistic fix for this problem involved testing the host's availability by using *ping*, before trying to *rsh*.
- The matter of timeouts. On rare occasions the *rsh* process would inexplicably hang during *execution*[7] which again staunched the flow of data.

  Taking the view that inexplicable "stoppages" of this sort would become more common and even less explicable as *AutoHack* reached out further into the WAN environment – into networks that were beyond our immediate administrative control – we decided that the logical solution to this problem was to wrap the probe processes with some form of "watcher", a program designed to kill the probe after a specified quantum of wall-clock time has expired.

  Generalising, this led to the creation of the *timeout* script, which launches a process specified on the command line and then allows the process to run for a specified period of time ranging from *seconds* to *days*, killing it, first with SIGTERM and then SIGKILL, when the period of time is over.

  This simple interface to robust timeout functionality was responsible for a sudden rush in the creation of probes based upon utilities found in the standard UNIX networking toolkit (some of which were otherwise too untrustworthy to be suitable), and *timeout* thus became the mainstay

[6] ...or ignorant. Either. It doesn't matter which...

[7] The prime suspect in our complex networked environment was that this was caused by a subtle interaction between *in.rshd*, NFS and *automounter*.

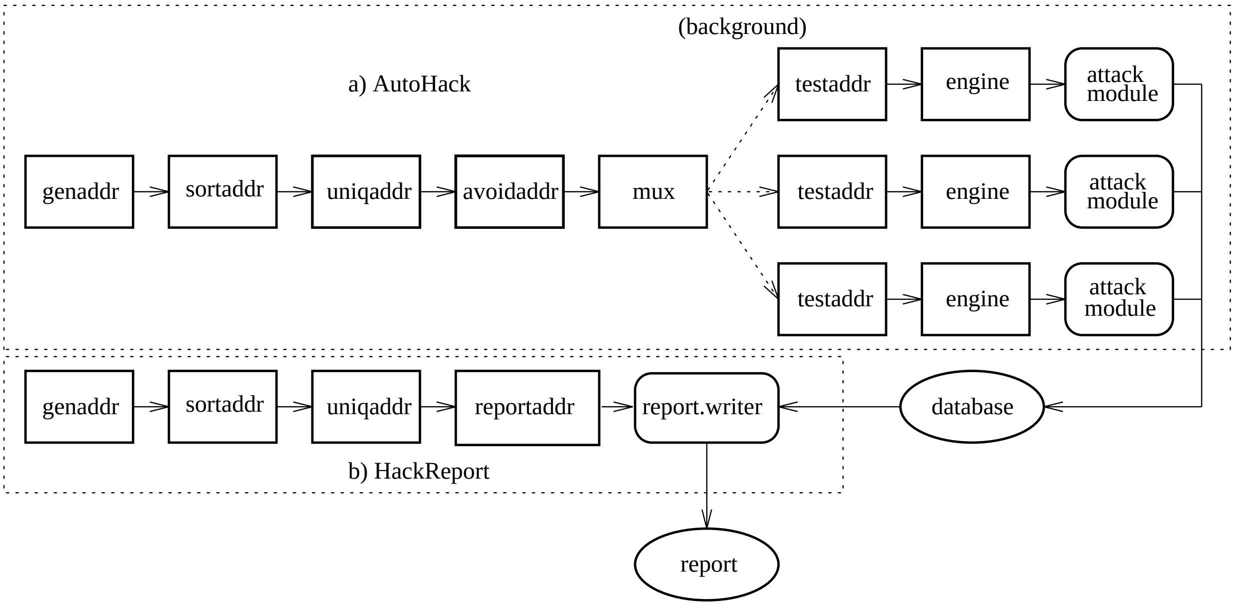

Figure 2: Anatomy of a) *AutoHack* v5.8 and b) *HackReport*.

of ensuring *AutoHack*'s reliability in the face of extreme system and network load.

Problems that we encountered only once we had begun probing the network, included:

- The output. Even with these extra enhancements, there is obviously so much *more* that could be done with this script, obtaining information about other services available on the target hosts

- The data collection mechanism *itself*. The fact that the data generated by this probe could only be reproduced by re-scanning the network – unless it was explicitly saved into a file by the user – placed an unacceptable strain upon the development. Repeating sweep after sweep of the local network was both irritating, and slow.

It was at about this time that *AutoHack* began to take a slightly different direction from the serial "attack and report" scheme used by other network scanners.

The original loop was modified to invoke a series of separate "probe" modules, specifying the target host's IP-address as the first argument. Timeouts were handled by the modules *themselves* so that the "driver" shell-script did not have to be modified when new probes with different runtimes were introduced.

The most important innovation, however, was a new way of indexing the output created by the probes. Results from each probe were stored as separate files in individual directories (referred to as "bins") with one "bin" for each host in the database.

Originally these bins were grouped together in a single flat `database` directory, and each bin was named after the host concerned, eg:

`database/mailhost.foo.com/`

References to textual hostnames were later replaced by IP-addresses, making access to the database "tree" less fraught with ambiguity, but eventually, as the `database` directory grew, experiment demonstrated that the most flexible and efficient naming convention for these bins was as a hierarchy reflecting the IP-address, for instance:

`database/127/0/0/1/`

...is the bin that would store reports for *localhost*.

Although this appears at first to be a unjustifiably expensive method of storing data, the benefits are immense: the database structure scales extremely well, permits fast access[8], simplifies incremental updating on a *per-host* or even a *per-probe* basis, obviates most

[8] A directory will never contain more than 257 links; typically only one quarter of that number of bins populate the third-level directories in the database.

file-locking or record-locking issues, and is easily implemented with tools from the standard UNIX toolkit.

With the new program structure came the opportunity to experiment; as the original shell-script was designed to read a list of target hostnames or IP-addresses from `stdin`, after experimentation it was found that a small but not negligible increase in throughput could be achieved by asynchronously testing the reachability of machines *before* attacking them, rather than pinging them synchronously before attacking. Thus was created *testaddr*, a simple front-end to the *ping* program, reading a stream of IP-addresses from `stdin` and filtering out all those which are not reachable.

The *AutoHack* script was now reduced to a simple argument-parsing shell-script which created a stream of IP-addresses, *ping*ed them in one process, and piped the live addresses into a hacking engine. This *engine* process in turn creates the database bins and launches the probes upon the addresses that it is supplied with, never[9] wasting time attacking "dead" machines.

This is the basic structure which is still used today (Figure 2), and which has endured some 5 major and 30 minor revisions. *Testaddr* itself has been changed slightly, and now consists of a single process which sends and receives UDP "ECHO" packets to test host reachability, saving the overhead of spawning a large number of individual *ping* processes, although *ping* functionality is also available if desired.

Further refinements to the pipeline included the introduction of *sortaddr* and *uniqaddr* filters which perform functions similar to their UNIX counterparts[10]. Given the central nature of a host's IP-address in the database, an intelligent *resolvaddr* module was written to canonicalise streams of IP-addresses and/or hostnames into a single format suitable for use by all the above modules.

The *avoidaddr* script also plugs into the pipeline to filter out spurious IP-addresses, and optionally to remove hosts which have already been logged in the database, or specific machines which you may wish not to probe.

Next came the driver module and multiplexer; it was no longer sufficient merely to feed munged `/etc/hosts` files into *AutoHack* as input – there was a pressing need to be able to exhaustively search IP-address spaces. *Genaddr* fills this need, taking a list of condensed IP-address patterns ("*ipaddrpats*") such as:

```
123.69.45-50.xxx
```

...either as command-line arguments or from `stdin`, producing a stream of raw IP-addresses as output.

With *genaddr* driving the pipeline, WAN-scale auditing became feasible, and made it possible to check every IP-connected machine on the network; however, in order to cope with this additional load, it became necessary to split the input amongst several processes. Thus the *mux* program was written to deal with this problem.

Similar in function to *xargs*, *mux* reads all of its input (taken from *avoidaddr*) into a temporary file and then splits the data into many equal-sized chunks (files), where the number of chunks can either be set by the user or automatically guessed at from the amount of data. *Mux* then spawns one *testaddr-engine* pipeline for every chunk, each one reading its portion of the work (as `stdin`) from the chunk-file created above.

This multiplexing allows the user to take much greater advantage of the available hardware, and reduces overall runtime significantly.

The Attack Modules.

The power of *AutoHack* lies in its structure and database format, but its usefulness comes from the probes (or "attack modules") that it uses.

To reiterate, each module is a single program which takes an IP-address as an argument, sets a timer to some sensible interval for execution, and performs a security-related probe upon the IP-address.

Modules are all stored in a single directory, scanned by the *engine* process on startup, and are named in this manner:

```
attack32.nfsserv
```

The "`attack`" keyword marks this program as being an attack module, as opposed to any other type of program. The number "`32`" refers to an attack scheduling mechanism similar to that used by System V's *init* program; attacks can be sequenced so that they can draw upon data retrieved by earlier probes.

The module's suffix "`nfsserv`" is the name of the file which will be created by *engine* to contain the probe's output. If this file is empty after the probe has exited, it will be deleted so as not to clutter the bin with useless data.

If this file is *not* empty, and a file (eg:)

```
exploit.nfsserv
```

[9] Almost.

[10] ...differing only by the fact that all comparisons are based upon numeric value of IP-address, as opposed to string comparisons.

```
# http probe
library lib.banter
tcp     123.69.42.7:80

# send an illegal command and log response
psend   BOING
call    flush_input
quit
```

Figure 3: *banter* code for probing HTTP daemons.

– exists in the `modules` directory, then this latter program will be launched at the host in a similar manner, in order to follow-up the probe's initial attack and to gather further information.

This technique of using sequenced attacks with automatic follow-up is useful when building complex probes which have special dependencies, and it also simplifies the creation of more advanced probes which can draw together output from earlier attacks and make inferences about the remote machine's security.

The evolution of the attacks themselves have shown the benefits of modular coding; in particular, the *rsh* attack carried over from the very first revision of *AutoHack* has been through three functional revisions with no modification whatsoever to the *engine*:

- su *locuser* -c rsh -n *hostname* echo *cookie*
- doas *locuser* rsh -n *address* echo *cookie*
- xrsh -verbose *address ruser luser* echo *cookie*

The first version calls upon *su* as "root" to forge credentials that would be used by *rsh* to test the ability to log in as (say) user "daemon".

The second revision utilised a custom perl script *doas* which provided similar functionality to *su*, except that it could cope with changing UID and GID to arbitrary values, or to users who do not have a valid login shell locally (eg: `/bin/sync` for user *sync*).

The third, current, and most powerful version of the probe uses a perl script called *xrsh*, which takes the r-protocol credentials supplied on the command line and passes them to the remote host as would any other *rsh* client, so that the probe can quickly test remote accessibility as any arbitrary username without requiring the same username to be installed on the local machine.

Xrsh also takes the notable step of specifically reporting whether the authentication credentials provided were accepted by the remote host, permitting *AutoHack* to distinguish between four states:

- the TCP connection was refused.
- credentials were rejected and the RSH connection refused.
- credentials were accepted and the command was executed.
- credentials were accepted and the command was not executed.

– the consequences of the first three possibilities are obvious, but the fourth is slightly more interesting; if a large number of accounts on a host accept the credentials but do not execute the command supplied by *xrsh*, it implies that the host has been "secured" by changing the login shell of system accounts to a non-standard shell (eg: `/bin/false`) without removing promiscuous trust from `/etc/hosts.equiv` – a situation which deserves further investigation and possible remedial work.

Probably the most diverse probes used by *AutoHack* are those which attack the ASCII-based TCP services such as SMTP (and its specific incarnations in programs such as *Sendmail*), FTP, NNTP, HTTP and FINGER – protocols which rely upon the exchange of ASCII text and/or standardised result codes for their correct operation.

After creating a couple of protocol-specific attack modules, it became evident that there was a great deal of code that would be replicated in all the modules of this type, increasing the overhead of code maintenance and multiplying code diversity within the suite – with all of the porting problems that this usually causes.

To alleviate this problem, all of the attack code pertinent to the "simple" TCP protocols was thrown away and replaced with code written in a custom assembly-like language, *banter*.

The *banter* interpreter is a 300 line *perl* script, providing primitives for functions, reusable libraries and program flow control, establishment of TCP connections to specified host and port combinations, timeout management, writing data to remote services, pattern matching of data received from remote services, symbol table management and basic debugging facilities.

Having abstracted this reusable functionality into a single, easily ported program, TCP probes become extremely simple to create. For instance, with transaction-oriented protocols such as HTTP or FINGER, there is little more to a *banter* script than sending a string to the remote server, and then reading whatever response is returned (Figure 3).

This simplicity has allowed us to proactively scan our network for particular network services, in case the information should become useful at a future time.

For instance: during a recent scare involving a particular WWW hypertext daemon, we quickly identified *all* of those machines which were at risk from the bug, by adding code to examine and report upon data that had already been collected in an earlier *AutoHack* run.

We had noted (some months previously) that many WWW hypertext daemons indicate their make and version number in the HTML document that they produce in response to an illegal request[BL93]. Partly because this sort of information is often interesting for its own sake, and partly because the probe was so easy to create (Figure 3), the HTTP probe was added to the suite on the "off-chance" that the data it collected might eventually prove useful. It did.

In a firefighting scenario, having this sort of information at your fingertips can be a refreshing change for most security personnel.

Much the same approach is taken with probing of other daemons, the prime objective being to eradicate buggy software – however, *banter* is not restricted to passive analysis of what information the daemon openly provides; more complex attacks can be created with only a little extra effort, for instance probing for deeply buried bugs in *Sendmail* (Figure 4) or in the file permissions of anonymous-FTP archives.

Of course, most of these attacks stand on the shoulders of simpler probes such as *tcpprobe*, a small but efficient scanner which reports active TCP port numbers on a specified host. Since the port scanner is always the first probe to be launched (`attack10.tcp`), all further TCP-based attacks can check the output of the scanner to ensure that the remote host really *does* support the service that would be attacked by the probe.

Many other probes rely on standard networking tools to gather their information; there is much that can be inferred from the output of *rpcinfo* and *showmount*, and of course simple tests such as trying to use *tftp* to steal a copy of `/etc/passwd`.

This is the nub of the argument above, that although the attack modules are important to *AutoHack*'s functioning, the real *power* of it, or any similar program, is that it can automate the centralised collection and analysis of freely-available (*publicly-available?*) data pertaining to an entire *network* of machines, and then provide a mechanism for filtering the merely "ordinary" results from other data which might hint that a host is suffering from poor configuration.

Much of this data is already accessible to anyone on the network with a standard operating system with standard tools like *rpcinfo*[11], but of course, not all of the services capable of broadcasting "publicly available" information are provided with widely-available user interfaces. It has been necessary in some cases to write code to probe and collect this data.

So it is with *ripprobe*, a small script which sends a RIP[Hed88] enquiry packet to a machine's *routed* (or similar) and receives in return a dump of the machine's routing table, which is then sorted by network metric and written to `stdout`.

This routing table dump can later be parsed by the report writer, and is useful for network mapping and for detecting unauthorised network connections and other anomalies.

Other problems are also probed, for instance ancient versions of *selection_svc*, *rexd*, and other poorly-authenticated services, and filestore exported globally through NFS. Like most other modules, these probes have evolved from the simplistic (checking to see if the particular service is registered with *portmapper*) to the concrete (exploiting the service to provide *evidence* that the problem exists).

These changes have been driven by the scale of the problem at hand; no-one will invest time and money into the eradication of a service from a network, when only 1% of the installed base suffers the security bug that you wish to eradicate. This explains *AutoHack*'s greater emphasis on bug *exploitation* compared to some other software – it is often necessary to supply administrators with concrete proof of the problem, so that they may make time to fix it.

The Report Writer.

The *AutoHack* report writer, *HackReport*, provides a dual to the *AutoHack* pipeline described above; many

[11] Since automating centralised collection can be as simple as writing a eight line shell-script (Figure 1) – the output of which can be scanned with *grep* – it seems unfair to refer to programs such as *Satan* and *AutoHack* as being intrinsically *threats* to network security.

```
        # stdlib and connect
        library lib.banter
        tcp     123.69.42.7:25

        # get the header, watch for continuation lines
        call    cfill_buffer

        # hi there!
        psend   HELO
        call    cfill_buffer

        # deliver one nearly-valid message
        psend   MAIL FROM: |
        call    cfill_buffer
        psend   RCPT TO: nobody
        call    cfill_buffer
        psend   DATA
        call    cfill_buffer
        psend   .
        call    cfill_buffer

        # try to send a viral message
        psend   MAIL FROM: daemon
        call    cfill_buffer
        psend   RCPT TO: | sed '1,/^$/d' | sh
        call    cfill_buffer
        skipt   ^[23]\d\d
        goto    smtp_abort
        psend   DATA
        call    cfill_buffer
        skipt   ^[23]\d\d
        goto    smtp_abort

        psend   dd if=/dev/null of=/tmp/AUTOHACKED

        psend   .
        call    cfill_buffer
        skipt   ^[23]\d\d
        goto    smtp_abort

        echo    autohack-5: suffers sendmail security hole #1

label   smtp_abort
        psend   QUIT
        call    cfill_buffer
        quit
```

Figure 4: Some *banter* code for probing *Sendmail*.

```
host 123.69.42.7 wibble
date Wed Jan 25 21:42:37 1995

  123.69.42.7    *****       rlogin: DIRECT ROOT ACCESS - root succeeded
  123.69.42.7    *****     sendmail: info - suffers sendmail security hole #1
  123.69.42.7    ****   hosts.equiv: netgroup in hosts.equiv - +
  123.69.42.7    ****   mail daemon: info - there is a decode alias !
  123.69.42.7    ****        rlogin: DIRECT RLOGIN - as bin succeeded
  123.69.42.7    ****        rlogin: DIRECT RLOGIN - as daemon succeeded
  123.69.42.7    ****        rlogin: DIRECT RLOGIN - as guest succeeded
  123.69.42.7    ****        rlogin: DIRECT RLOGIN - as sys succeeded
  123.69.42.7    ****        rlogin: DIRECT RLOGIN - as uucp succeeded
  123.69.42.7    ***     ftp daemon: info - anonymous ftp is enabled and works!
  123.69.42.7    ***     ftp daemon: info - root ftp is enabled
  123.69.42.7    ***    mail daemon: info - postmaster mail goes to bitbucket
  123.69.42.7    ***         rlogin: connect - as sync permitted
  123.69.42.7    **        /.rhosts: obtained - bong root
  123.69.42.7    **      domainname: obtained - nis
  123.69.42.7    **     hosts.equiv: obtained - +
  123.69.42.7    **         tcp svc: info - 110 [pop]
  123.69.42.7    **         tcp svc: info - 80 [www]
  123.69.42.7    **           uname: obtained - SunOS wibble 4.1.5 42 sun4c
  123.69.42.7    *           routed: routes to - 123.23.21.0
```

Figure 5: Fictional output from *HackReport*'s report writer.

of the modules are reused, the difference being in the replacement of *engine* by *reportaddr*, a script which (like its counterpart) reads lines of IP-addresses from `stdin`, but then test for the existence of and changes directory into the appropriate "bin", and then (as opposed to an attack module) it launches a script called *report.writer*.

The *report.writer* script has a single task: to test for the existence of files that have been left behind by the attack modules, to parse their contents, and then to summarise its findings in a comprehensible manner (Figure 5). The script's task is eased by the way that the *banter* scripts have been written; many scripts produce "cookies" in their output – small, specially-formatted strings which reflect the problem that *banter* has detected – and these cookies are picked up by *report.writer*, reformatted, and included in the final report.

As an aid to comprehension, all of the notable facts reported by the default *report.writer* script are associated with a arbitrary severity rating – a string of between zero and five "stars", where trivial facts about the host are rated as one-star, important information and configurational issues are three-star, and direct root access (or similarly nasty bugs) are rated as five-star.

This simple grading allows administrators to see at a glance what problems need to be dealt with on their network, and has yet to be misunderstood by anyone to whom it has been explained[12].

We have examined the possibility of creating a modular report writing system in the manner of the *engine* program, permitting "drop-in" report modules to be tied to the functionality of their respective attack modules; however (although the idea has not been entirely dismissed) experimentation has shown that the extra time imposed upon the report-writing process by the overhead of spawning separate report modules would be unacceptable, and would impose other problems in (eg:) the ordering of security "facts" by severity.

Experiences and Feedback.

The *AutoHack* software suite runs successfully on both Solaris 1.x and 2.x, and has been tested on Slackware Linux 2.1, upon which it partly fails, due to a feature of the TCP stack in Linux kernels up to at least version 1.2.5.

AutoHack's throughput depends upon the ability

[12] Who says that security tools need GUIs ? 8-)

to rapidly create and destroy processes, files and network connections. Because each engine typically occupies 3 to 5 process-table slots – each engine launching a variety of short-lived processes – to run (eg:) 20 simultaneous engines requires enough physical memory to comfortably sustain 100 processes[13].

In a restricted setting, *AutoHack*'s resource requirements can be configured to be quite low (with a corresponding impact on throughput), but for preference it should be run on a machine with a moderately large quantity of of both physical and virtual memory so that paging is kept to a minimum, also providing sufficient room for those transient processes which may grow to be really large[14], although this is rare in normal usage. Heavy paging activity severely abraids the performance of *engine* processes.

As part of the automatic load-balancing scheme, and as an aid to portability, the *mux* process makes an empirical guess at how many *engine* processes should be launched, based upon the total number of IP-addresses that are to be scanned – on the presumption that only "well-equipped" machines will be set to probing networks of many thousands of hosts.

In extraordinary circumstances, the user may override this value by changing a variable in the driver script, but otherwise this feature permits *AutoHack* to behave sensibly when tasked with attacking any number of IP-addresses – from networks linking a handful of "secure" machines on a firewall DMZ, to networks of several hundred thousand potentially Internet-connected hosts.

With regard to efficiency, the standard architecture of *AutoHack* (Figure 2) has proven to be very effective when the list of IP-addresses created by *genaddr* is highly populated with "live" machines, but it is less than optimal in "sparse" address spaces, as is common when scanning all possible addresses in a particular set of subnets.

In such cases it is not unusual for an exhaustive list of IP-addresses to be split into a dozen equal-sized chunks and fed to hacking engines, one of which may complete in a few hours after attacking the few "live" addresses which were supplied with amongst its input, whilst another engine might take several days to complete because nearly every address that it was supplied with was "live".

This obviously is not the most efficient use of the resources at hand, which would occur when all engines completed their work at approximately the same time.

An alternative to this method could be to generate the list of IP-addresses (using *genaddr*) and test their availability (using *testaddr*) before passing them in a *round-robin* fashion to one of several concurrent *engines*, via some form of asynchronous multiplexer similar to a real-time version of *mux*.

Although this latter strategy appears better in many ways (faster completion of attack, even distribution of workload amongst the attack engines) its disadvantages include buffering issues within the multiplexer, bottlenecking of *testaddr*, and concentration of network load into "hot-spots".

The latter point bears some explanation: in the current architecture, sorting the IP-addresses before splitting the list into serially-probed "chunks" has the effect of evenly distributing the network load created by the attack engines, around the WAN.

Because similarity of IP-address in a WAN environment usually reflects geographical proximity of the machines in question, and therefore reflects the likelihood that all IP-addresses in a particular chunk are served by a single long-haul link, it is sensible to arrange matters so that only one *engine* is loading that link, to avoid network congestion.

The existing strategy exhibits this desirable property without any special treatment; the proposed one risks precisely the opposite – if presented with sorted input, several engines will attack hosts over the same long-haul network link simultaneously, causing both network congestion and ill-will.

Future research may find a way (short of randomising the order of input) to overcome this problem, increasing performance by an estimated factor of two – but for the moment the preferred solution to the efficiency problem is to create a new *AutoHack* database in two stages; first using a list of hosts which are likely to exist and be "alive" (eg: from DNS) and then second, against an exhaustive list of all possible IP-addresses *except* those which have been put into the database during the first stage.

In this manner, all engines (during each run) require approximately the same amount of time to run to completion, and the resources of both the machine and the network are used to near-optimal efficiency.

Meaningful statistics regarding *AutoHack*'s performance are hard to generate, and even harder to explain without causing confusion. The following empirical values are offered in lieu of hard benchmark figures; because they reflect the specific circumstances under which *AutoHack* is being run within Sun – network infrastructure and resources available on the host platform – they provide little more than hints as to *AutoHack*'s capabilities.

[13] When running at full speed, it has been noted that *AutoHack* can easily orbit the process table in 2..3 hours.

[14] eg: *genaddr* has been seen to occupy 34Mb of virtual memory in extreme circumstances; this is believed to be a side effect of the way memory is sometimes (not) reused in Perl 4.036

target network – *AutoHack*'s target network is a TCP/IP Internet comprising several Class B IP-networks which yield a total of some 1200 subnets and an address space of some 305,000 potentially "live" IP-addresses.

Of these addresses, some 30,000 are "live" interfaces attached to hosts scattered around the world, interconnected by a variety of networking technologies of differing bandwidths, typically in the 64Kbit to 256Kbit range.

Most notable amongst these are a pair of 128Kbit trans-atlantic links providing connectivity between Northern Europe, the North American continent, and (eventually) the Pacific Rim.

Given that the *AutoHack* project is developed and run from an office in the United Kingdom, and that a major portion of the address space will be accessed across these connections, these links are particularly interesting.

time expended – *AutoHack* completes a two-part scan of all 305,000 possible IP-addresses and 30,000 hosts in a little under 8 days, using 12 *engine* processes on a 32Mb SPARCStation 10/40 running Solaris 2.4[15].

In comparison, in a single-step *exhaustive* search of the target network, the 12 *engine* processes will terminate independently of each other, with the first typically exiting after 4..5 days, the last after 9..10 days.

This "window" of completion is due to the sparse nature of the address space in these circumstances (a host density of about 10%), and the sub-optimal use of resources as described above.

per-attack bandwidth – An attack on an individual host typically generates a total of between 30Kb and 80Kb of bi-directional traffic at the IP layer, depending upon the success and depth of the attack, the nature of the networks linking the local and remote hosts, network load, etc. An attack on an individual host typically requires between 90 seconds and 4 minutes to complete, similarly constrained by the state of the network.

Attempting to convert this data into a meaningful figure of long-haul bandwidth occupancy is tricky, since other factors such as fragmentation, network speed, and latency must be taken into account; therefore we shall merely note that this is a small but non-negligible amount of traffic, and thus is a powerful argument for *not* running large numbers of probes simultaneously across a single long-haul link.

bandwidth observations – Approximately two-thirds of the total address space is probed over the two transatlantic links mentioned above. Over the eight day period, the traffic generated by these probes peaks at occupying 15% of the total bandwidth available on each link, and more typical occupancy figures are between 5% and 10% of the total bandwidth available.

This is satisfactory enough to keep our network management team happy[16], and to favour maintaining the centralised nature of the *AutoHack* software and database, as opposed to distributing it over the network.

database size – The database storing the probe results for the above network occupies approximately 320Mb. The directory structure of the database accounts for approximately 15Mb of this space. This was surprising; intuition led us to believe that the overhead of "all that directory structure" would be an enormous factor in the eventual size of the database. As it is, the directory structure occupies perhaps 5% of the total.

user detections – *AutoHack*'s reported detection rate was quite low in the early days, estimated to be about 1 detection for every 500 hosts attacked.

This can be explained through a combination of influences, involving an administration culture that is "safe, behind the firewall" and the related phenomenon of "blithe trust".

Another reason for the initial low detection rate may be that *AutoHack* was initially designed to tread lightly upon hosts and upon the network, probing only those services which would generate little or no audit trail in a system. Hosts which were equipped with *TCP Wrappers*[Ven92] or similar fared well in the detection stakes, so long as the user checked the logs frequently; several "detections" were reported some *weeks* after the fact.

Once *AutoHack* was firmly established as a project, however, the probes (especially those relating to *Sendmail*) became considerably less "quiet". It is not now possible to specify a meaningful detection rate, because reports arrive in-

[15] Only 12 *engines* are used, in order to leave some CPU-time free for other tasks; *AutoHack* by default would use 16 *engines*, which would just about tie up the CPU, and certainly hamper interactive use of the machine in question.

[16] ...or, as they were initially, ignorant...

frequently, and only from users who have not encountered *AutoHack*'s activities before.

One particularly vexing problem with the strategy of sanctioned and vigorous auditing is the risk of "crying wolf", the fear that systems administrators will become desensitised by the assaults that *AutoHack* makes upon their machines, and that a host could then be configured to masquerade as the well-known *AutoHack* machine, and could rove freely around the network, attacking *everything* whilst being ignored by *everyone*.

The only solution that we have yet found for this problem is to promote an adversarial-but-friendly attitude amongst the systems administrators, congratulating them upon detecting probes by the real *AutoHack*, and encouraging them to report their detections back to us, with comment if they so wish.

This solution is both cheap and quite popular, because it opens up opportunities for education and promotes general interest in security issues throughout the company, whilst providing us with feedback about *AutoHack*, and yielding the data necessary to detect unsanctioned probes.

On another topic related to "noisiness", it is perhaps worthwhile noting that one function *AutoHack* does *not* currently perform is that of NIS domainname-guessing and subsequent theft of password files.

This has not been been implemented yet chiefly because within a corporate organisation there is no need to *guess* domainnames – after all, you can just ask the administrator concerned, where necessary – and moreover it was felt that checking passwords belonged more to the field of *host* auditing, rather than *network* auditing.

The problem that a NIS password map can be stolen remotely (using *ypx* or similar) is a facet of a wider problem – that of RPC authentication – and should be dealt with as such. This particular probe functionality may one day be added to *AutoHack* for the sake of completeness, but for now the problem is being addressed in a *holistic* manner.

Conclusions.

AutoHack is far from complete as a security tool – much could be done to it in terms of performance: enhancing its probing ability, speeding it up with enhanced versions of the filters that it already uses, making use of asynchronous I/O rather than simply-buffered pipes for inter-process communications, etc, but there are also new features being discussed which should be part of future versions; for instance, addition of UDP datagram support and `pty` (pseudo-terminal) management to *banter* would enhance *AutoHack*'s probing capabilities immensely.

Obvious structural improvements include the implementation of some form of history mechanism to automatically mark as "high priority" any security hole which remained unfixed for an extended period of time, so that escalation reports can be generated and passed directly to senior management who have an interest in security.

This is probably most simply effected by front-ending the *report.writer* script with the history mechanism, but since this runs against the blithe "throw it away when you don't need it anymore" mindset behind *AutoHack*'s design (the history mechanism requiring a reduced copy of the database to be kept for an extended period of time) – the implementation of this feature is under *very* careful consideration.

An interesting comment was made by a member of our networking team; he noted that there did not appear to be a readily-available dual to the *TCP Wrappers* suite for detecting suspicious protocol traffic as it *passes along* network backbone.

Network monitoring software that we currently use is keyed towards detection of suspiciously high traffic loads, or for watching for traffic being sent to or from unregistered subnets and illegal IP-addresses (commonly caused by misconfigured hosts). Nothing in the software we had was designed to trigger an alarm upon detection of unusual protocols on the wire.

Creation of a tool for this purpose may be an interesting project for someone so inclined, perhaps a program designed to learn the profile of normal network usage as described by *tcpdump* or similar, with the knowledge engine reporting anomalies in real time.

What lessons have we learned?

If nothing else, the experience of *AutoHack* has borne out several old prejudices: "standardisation" is both a pain and a panacea in computer security. Where bugs exist in "standardised" machines, they are rife, because an error in one configuration or security policy is propagated to many other hosts, either by wholesale duplication of the host's software, or identical installation methods.

On the other hand, administrative tribes who take "standardisation" seriously are usually well-equipped to deal with the roll-out of a security patch across all of their hosts. Users who run their own systems tend to be slower to fix holes unless they are presented with unequivocal evidence of the bug's existence – something that the *AutoHack* database is well equipped to do for them.

The one overpowering lesson, however, from the creation of *AutoHack* and the response it has generated, is this: network security does not evolve, either in terms of the security of the installed base of hosts in a network, or in terms of software development, *except in a hostile environment.*

Only in the presence of a threat to security – a clear, present, and well-advertised threat, less shadowy than "hackers", more definite than "the potential for viruses" – will people *act* in order to improve their security.

If the threat can be not merely be contained so as not to cause malicious damage, but can further be controlled so as to "inoculate" the network, so much the better.

AutoHack is by design a benign but definite threat, and it serves this purpose well.

Availability.

At the time of writing, *AutoHack* is only available for use within Sun Microsystems Computer Company, to audit its internal network.

References

[BL93] Tim Berners-Lee. Hypertext Transfer Protocol. *ftp://ftp.w3.org/pub/www/doc/http-spec.txt*, 1993.

[CB94] William Cheswick and Steven Bellovin. *Firewalls and Internet Security.* Addison Wesley, 1994.

[Far94] Dan Farmer. personal communicaton, 1994.

[FV92] Dan Farmer and Wietse Venema. *Improving the Security of your UNIX system by breaking into it*, 1992.

[Hed88] Charles Hedrick. Routing Information Protocol. RFC-1058, 1988.

[Ran93] Marcus J. Ranum. Thinking about firewalls. In *Proceedings of the Second International Conference on Systems and Network Security and Management (SANS-II)*, 1993.

[Ven92] Wietsa Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the third USENIX Unix Security Symposium*, 1992.

Most of the software cited in this paper may be retrieved from the COAST computer security archive:

FTP: `ftp.coast.cs.purdue.edu`

WWW: *http://www.cs.purdue.edu/coast/coast.html*

Acknowledgments, etc...

The author would like to thank Brad Powell for his long term help in acting as a sounding-board for ideas related to development of *AutoHack* and this paper, and is grateful to Chris Samuel and Simon Halsall of D.R.A. for their exceedingly useful review work. Many thanks also to Marcus Ranum for saying that the topic sounded interesting enough to be worth writing up.

Finally, many thanks to Gillian Anderson for (usually) letting the author get away with all of those late nights associated with the preparation of this document.

Alec Muffett lives near Oxford and works for Sun as a member of the *Network Security Group*, responsible for policing and auditing Sun's internal network, evaluating security products and architectures, and incident handling.