# How to Make Sure That Nobody Will Ever Use My Excellent Software (Twice)

Benedikt Stockebrand

EuroBSDCon 2006
Milano, November 12, 2006

**Abstract**

One of the greatest successes that a coder can achieve is writing a piece of excellent software and at the same time ensuring that nobody will ever use it—or nobody will ever use it twice.

This talk presents a number of real-world highlights that show how minimum effort can achieve maximum devastation, teaching even the most stubborn user or administrator a lesson it deserves—a lesson to stay away from my software.

## Understanding the Enemy

As in any battle the first step to victory is a thorough understanding of the enemy. For our purposes it pays to distinguish four different subspecies: The end user, help desk staff, administrator and management; they all have different vulnerabilities and should therefore be treated specifically.

### The End User

The most clueless victim is the end user. It has the least possible grasp of technology and notoriously refuses to read documentation of even error messages, usually either complaining of "techno babble" or pointing out that it doesn't understand English documentation. The end user comes in two varieties: Private and business end user.

The private end user is easy to handle; if we just frustrate it enough it gives up using our software quite quickly.

The business end user is usually told by its management what to do and can't just give up on our software. Dealing with it directly can be fun because it can't really defend itself, but usually we can only teach it to be terrified of our software, but can't really make it stop using it.

### The Helpdesker

The business end user is usually supported by a help desk team. The helpdesker usually has a better grasp of the English language, sometimes enabling it to read and understand end user documentation. It is technically marginally clueful and tends to gather quite a bit of experience from dealing with all the problems its end user has.

Like the end user, the helpdesker generally doesn't have much of a choice of the software it has to support. Unlike the end user, the helpdesker is a notorious job hopper, so it is feasible to convince it to hop its job even faster than usual if we write our software accordingly.

### The System/Network Administrator

Beyond the help desk hides the system and network administrator. It is at least semi-clued; in some cases it may even have developed basic coding skills. It tends to understand even technical documentation, even if it is written in English.

The admin is usually overworked and has to deal with a wide range of systems. It usually doesn't have time to get into all the details of all the systems it take care of and therefore spends a ridiculous amount of time searching documentation over and over again for details it needs to solve the problem at hand.

A major part of its working hours the admin spends troubleshooting. This gives it an enormous routine to stay cool even in an "exceptional failure" situation—it is just daily business to the admin. A good admin works with great care even in a major emergency situation where every minute costs dearly (but bad admins won't).

At least the senior admin may occasionally be trusted to decide which software to use by its management. So at least in some cases we must consider it a direct target throughout our efforts to prevent it from using our software.

### Management

The ultimate target however is management. The Manager isn't even proper business end user by qualification because it has its underlings to do the business end using. But it is the one who is "in charge" and in many cases it will decide for an entire company to stay away from our software.

This is a blessing for us. Since the manager doesn't have the technical competence to realize that our software is excellent, we can achieve our goal without sacrificing any of the excellence of our software.

The only problem with management is that it is hard to reach. But once it gets alarmed it is bound to make some heavy-handed decisions against using our software, no matter what its technical qualities are.

## The Software Lifecycle: A Monetary View

The key strategy to deterring the prospective enemy from our software is hitting it where it hurts it most: at its wallet. If we hit hard enough and at the right moment, then we can't but succeed in our mission.

Hitting hard is a matter of the tactical weapon we use; we'll take a look at our armory in the next section. But hitting in the right moment is even more important than the way how we hit. This leads to a brand new sort of software lifecycle, one that focuses on what happens when the software gets shipped to the enemy and how it relates to its money.

On a side note: There is a common strategy pursued by certain "consulting companies" which "do the IT projects" for an IT-illiterate customer. Then run up such a huge bill that the customer's management can't possibly admit to the shareholding public that the "consulting company" delivered an entirely useless heap of junk, so they declare the project a "success" for "political reasons". The result is almost exactly the opposite of what we want to achieve: They make their customers use disastrous software, while our intention is that we want to make the enemy *not* use our *excellent* software. Both strategies do have one thing in common, though: Both will drive the admin crazy and quite likely away from that company.

### Evaluation

When a data center intends to install new software, it usually starts with an evaluation of the various alternatives available.

At this point it is generally easy to deter our prospective enemy: Bad documentation and possibly some wanton flaming in the support mailing lists explicitly set up for the new enemy usually do the trick.

At this point it is also surprisingly difficult to deter the prospective enemy in the long run: There is little money and time involved yet, so the enemy may decide to give our software another try later on.

### Installation and Configuration

At the end of the evaluation phase the enemy chooses a software. During the following installation and configuration phase we can apply a variety of attacks.

But generally this is not the best time for an attack since there is still too little money involved. Instead we should keep a low profile, try to appear helpful and give the enemy the impression that there won't be any problems with our software. The installation and configuration phase is the last chance for the enemy to retreat without major losses.

Only if the installation and configuration phase is managed as part of a software project with the customarily infeasible deadlines, then some stealthy tactics to delay the installation and configuration may be worth a thought.

### Production

After the initial installation and configuration phase the enemy finally places itself at our mercy.

Sometimes it realizes this and tries to negotiate its surrender through a "pilot phase" that it doesn't consider regular operations. Often the administrator is particularly vigilant when then it puts a system into production. But hurting the enemy is primarily about hitting its wallet, so at this point the enemy can't possibly escape our wrath any longer.

There are three basic attack lines relevant to production systems: We can make regular operations excessively expensive, create havoc during upgrades including security patches, and lay some fatal traps that only bite the enemy when it already struggles with another problem.

### Regular Operations

To attack the enemy during regular operations we must make operations as expensive as possible. If we make the software difficult to use, then the end user is the first to give up, causing excessive workload to the helpdesker which needs additional manpower which costs the management additional money.

Similarly, we can make administration more expensive than necessary. Any software that makes daily operation a tedious routine job or requires extensive knowledge and experience in widely different and unrelated areas will be considered an excessive expense by our involuntary accomplice in management, which calls itself "controlling". While this may sound fairly unexciting to a software developer, the sums involved can be large enough to drive the enemy manager to despair and therefore away from our software.

Another useful strategy related to regular operations are excessive requirements for service downtimes. While these can be scheduled in advance, they still put a burden on many installations. Again, the financial losses during downtimes can be substantial enough to convince th enemy manager to convert to an inferior software.

Finally, problem hiding is a way to drive the enemy to despair: If it never really knows if everything works as expected or if some problem is quietly creating some yet unseen havoc, then it will quickly decide that it doesn't want to use our software anymore: The admin is permanently stressed from worrying if everything works and the manager considers the software as well as the admin unreliable, which is reason enough for either one to look for an alternative to our software.

## Upgrades

The enemy is even more vulnerable during upgrades. Of course it knows about this and avoids upgrades whenever possible. But if growing systems, increasing reliability requirements, exciting new features, support for new hardware or ending support for older software versions don't convince the enemy to face the upgrade, then an exploitable security hole will.

The only defense it has is extensive preproduction testing. A reasonably experienced enemy will avoid haphazard upgrades and rather spend time and money on preliminary tests. But duplicating an entire datacenter environment doubles the expenses, so in many cases the enemy only has a limited test environment used for multiple software installations in turn. So a pre-upgrade test works like this: Somebody decides that an upgrade is necessary, then a time slot for the test environment is allocated, a large number of tests are done in short time and afterwards the enemy will still worry if it missed the one crucial thing to test that'll blow up in its face. If we don't leave it a chance to revert to the old version this situation will make the enemy grow old way before its time.

With desktop machines the situation is different: A test environment doesn't need as many desktop machines as there are desktop machines in use, which saves some money on the test hardware. But rolling out an upgrade to several thousand machines running a large variety of applications is quite similarly scary.

The obvious strategy of throwing a monthly batch of updates plus the occasional "extra urgent" security fix at the enemy is well-known and has proven useful over and again. It does however require an unacceptably low level of software excellence to be applicable, so it is less useful to deter the enemy from using our excellent software.

Still, there are other means to send the enemy to the upgrade hell; we'll see examples below.

## System Failure

Finally, there is the ultimate victory scenario, the one scenario that strikes the very heart of the enemy.

The new junior admin, which was just hired three weeks ago, is on weekend call duty for the first time. On Sunday morning, 03:00, its mobile phone rings. The admin answers, only to hear "machine $XYZ$ doesn't work, come here and fix it immediately". Of course, it has never ever heard of machine $XYZ$, but it drives to the data only to be welcomed by an unscheduled meeting of the management board with the words: "Do you actually realize that every minute of this costs us 50 000 €?"

Of course, the very same scenario applies to senior admins in large enterprises if the 50 000 € are substituted by a more adequate number. The sky is the limit here: Occasional rumors claim that the Deutsche Bank will be bankrupt within 24 hours if their entire IT breaks down. For the recent two-hour failure of the Spanish top level DNS domain no numbers seem to be available, but at a national economic scale the losses caused by this might just possibly exceed 50 000 €/minute, or 3 000 000 €/hour—by a few orders of magnitude.

This scenario is so valuable for a variety of reasons. Of course, the immediate economic impact is obvious: we hurt the enemy where it hurts most—at its wallet. Besides that, we score multiple severe psychological hits: The end user gets upset because it can't work. The helpdesker gets upset because it has to answer the phone calls of the end user while it doesn't really know what's going on. The admin feels seriously embarrassed because it appears incapable of keeping its systems up and running. The manager feels helpless because it has no idea what's going on or how long it will take to fix, but a reasonably precise idea of the huge losses per minute.

Even in situations far less dramatic than those mentioned it is quite simple to make our software so expensive to use that the enemy will eventually give up and leave our software alone; we just need to

make system failures expensive. We can make failures happen often, and we can make them last long. Beyond that we can make it impossible to repair the system completely by introducing inconsistencies during the failure; this will make it necessary to roll back to the last working backup, losing all data changes since then. All three approaches can be combined and put enough economic pressure on the enemy to convince it quickly to use some other, less excellent software than ours.

If we want to keep the level of excellence of our software, we can't resort to causing the problem within our software; instead we must ensure that the "cause" of the problem lies outside, usually either within the system environment or the user or admin operating it.

# Masterpieces of Deterrence

Now that we understand the enemy and its vulnerabilities, we can understand and assess the various tactics available to us. After so much theory we now follow a more practical approach and take a closer look at successful examples of enemy deterrence.

## Documentation

Documentation-based deterrence measures are quite common, often because documentation isn't really considered part of the software and thus exempt from the goal of excellence. But even if we consider documentation part of the software, there are some excellent tactics available.

The most obvious tactic related to documentation is used by various low-budget hardware vendors: Write the documentation in whatever language we are unfamiliar with, then run it through babelfish to translate into Albanian, and then have a native Chinese speaker translate the result into English. As long as we don't consider documentation part of our software, this approach is generally known to work as expected. Unfortunately, the enemy will quite likely notice this during evaluation, so the impact is very limited.

A more effective, diametrally different approach involves a native English speaker linguist specializing in classic English literature polishing the documentation to unsurpassed beauty: While the language of Shakespeare, Melville and Thoreau may be most elegant and stylish, it is impossible for a non-native speaker with a more limited grasp of English to understand any of this.

The Debian[1] project came up with a nice way to deal with missing documentation: There is an `undocumented(7)` man page that the package maintainers generously use as a substitute for non-existing man pages. The subtle psychological effect of this is quite remarkable: The enemy will almost invariably interpret this helpful note as "I know I let you down; sue me."

Beyond that, Debian Woody made generous use of the Linux-specific `ip` command to configure its network, rather than the `ifconfig`, `route`, `arp` and various other commands commonly used with Unix. The documentation available came in three variants: LaTeX source code with more than 80 chars per line to make it less readable on a text console, DVI intermediate output and PostScript. The subtlety of this is brilliant: During the evaluation, installation and regular operation it is quite likely that the enemy will simply use this documentation. Only when a problem occurs that drops the enemy into text mode only will it realize that it can't any longer read the documentation it desperately needs.

FreeBSD 6.1 installs with a file `IMPLEMENTATION` in `/usr/share/doc/IPv6` which states right at the beginning that it doesn't relate to the KAME IPv6 stack integrated with NetBSD 1.5.1, but might still be useful. Following that the table of contents has an entry "7.2 Multipath Routing Support". Except of course that chapter 7 covers coding style and there is no section 7.2. So if the enemy doesn't bother to test for multipath routing support during the evaluation period but relies on the assumption that it can use multipath routing later on because it was mentioned in the table of contents, then the enemy will stumble over this only when it tries to use multipath routing when the system is hopefully already in production.

The man page for `dig` with FreeBSD 6.1 shows a date of June 30, 2000. During the last six years a variety of changes to the `dig` command have found their way into the source code. It takes a very close look to realize that only the date hasn't been updated in the man page but everything else has. In a high stress situation like troubleshooting this tiny little lapse will easily extend the downtime by several minutes; the average enemy admin tends to be overly careful when working in production machines and won't really trust this man page until it has verified that its contents is actually up to date.

The man page for `cvs` starts with this note: "This man page is a summary of some of the features of `cvs` but it may no longer be kept up-to-date. For more current and in-depth documentation, please consult the Cederqvist manual (via the `info cvs` command or otherwise, as described in the SEE ALSO section of this man page)." Again this is a gem: During evaluation and installation the introductory style, texinfo based manual is generally preferable over a man page. But when a problem occurs, then a more concise, reference-style

[1] OK, "Debian GNU/Linux", if you insist.

man page is needed, not a lengthy tutorial-style info file. And we have shown all the goodwill the enemy could ask for: We have provided a reference man page as well—if it assumes that the man page is unreliable, well, that's its own decision.

## Support

Similar to documentation, support is often not considered part of the software proper and therefore lends itself well to various tactics. Some of them are more commonly seen with commercial software vendors, but they are still inspiring enough to be mentioned.

It is common to discontinue support for old software versions as soon as possible. Together with promises of new features, this is a common way to lure the enemy into upgrade hell and no particular surprise. Several years ago SUN support vastly improved this tactic: Whenever the enemy tried to open a problem call they would first demand that it updated the system with the latest "recommended patches". With this strategy they managed to force the enemy to deal with a system failure while in upgrade hell at the same time. Unfortunately, the enemy eventually realized developed a counter-strategy of demanding a written guarantee that the new, untested "recommended patches" wouldn't affect the system adversely but actually helped to solve the problem at hand.

Open source projects offer another line of tactics: Since support usually isn't paid for, it is easily possible to repel the enemy by simply telling the truth: "If you can't read the source and figure it out yourself, then get lost. I can't be bothered to write documentation and I definitely can't be bothered to tell anyone how to use my excellent software." If we do this only after the enemy has reached the problem-solving state, then a few well-chosen insults will quickly deter it from using our software once and forever.

## Wanton Limitations

Another set of tactics relate to imposing wanton limitations on the software or the system it runs on.

Limitations that relate to the software itself are usually a sign either of bad software or unscrupulous money-making: The various size limits for IDE hard disks imposed on various generations of BIOSes are all signs of bad software design; selling different kinds of system phones to different PBX systems has been used by phone manufacturers to force the enemy to replace not only the PBX but also all the phones in a company as soon as the company grew beyond the capacity of the old PBX. Either way, while these tactics are proven to be effective they can't be applied to excellent software too easily.

A more useful and less conspicuous way to impose artificial limitations involves interference with other software. Back in the Good Old Days[TM] it was impossible to install both what later became FreeBSD/NetBSD and DOS on the same hard disk, simply because BSD didn't support PC-style partition tables. Since this will usually be noticed during evaluation, the impact on the enemy isn't too exciting, but it quite effectively scares it away. Microsoft has refined this to the "Windows DLL hell": If different programs need different versions of a certain dynamically linked library (DLL, the Windows equivalent of a shared library), then they can't run on the same machine.

## Excessive Dependencies and the Autoconf Trick

Even more useful and less conspicuous are excessive dependencies.

Solaris 10 offers "zones", the Solaris equivalent of FreeBSD jails (but with IPv6 support). These zones can't be used without installing resource pools, which need a Java runtime, which need X11 even on a server with a serial console and no video hardware.

Again, this can be improved. The autoconf-generated configure scripts commonly used by open source projects can be easily used to create excessive dependencies that we can blame the enemy for: If it finds KDE, why not use it? And GNOME too, and a Java runtime and SSL and SNMP and an OpenOffice programming interface. Now if the enemy builds our software on its desktop machine, then it has no choice but to install all these things on the final destination machine, too. We can't possibly be blamed because our software doesn't really "depend" on these dependencies, it just makes use of them if they are available anyway. And next time the enemy builds our software again, chances are that it has installed yet some more software on its desktop machine and if it installs the newly built software on the destination machine it will fail because the enemy hasn't installed the additional software there, yet. Beautiful.

All these tricks also open the path for additional fun with respect to security: If our software uses some insecure dependency, like the intrinsically insecure SNMP, or anything with a less-than-impressive security history, like a certain commercial web browser, then we can use these dependencies to force the enemy into upgrade hell more often than it can handle.

## Configurability

To make software excellent we must make it configurable to the enemies needs. So if we want to preserve the excellence of our software, we can't use configurability as a weapon against the enemy—or can we?

The Asterisk IP telephony software offers a highly configurable "dialplan" configuration which defines the behaviour of the software. Its lines must be numbered like ancient BASIC programs. Unlike BASIC programs the line numbers must be consecutive. Errors cause a jump to the current line number plus 101, so if an error in line 7 occurs, line 108 will be executed next. A local Asterisk expert summarized this in this way: "If you want to write a dialplan, then do it in a single day. If you don't, then on the second day you won't understand what you have done the day before and start from scratch." This makes any configuration change later on a nightmare to the enemy; the first time a change is necessary, it takes all day and breaks some previously functional features will quite likely encourage the enemy to look for an alternative solution.

Another example is the traditional `sendmail.cf` to configure Sendmail. Besides the syntax, which is annoyingly difficult to understand, a Sendmail configuration has to be fairly large. Writing a `sendmail.cf` from scratch is quite demanding and requires detailed understanding of the SMTP protocol to keep the result standards-compliant. Unfortunately somebody made the huge mistake to write an m4 macro package that generates a `sendmail.cf` from a fairly short and readable configuration file.

The traditional `sendmail.cf` offered an excessive degree of configurability that virtually nobody needs anymore today. DECNET, BITNET and UUCP are effectively dead, but still Sendmail has everything necessary to support them. Again, if it wasn't for the m4 macros this would serve quite well to make the enemy switch to Microsoft Exchange without looking back.

Even with just a few configurable parameters it is possible to make configuration tedious and error-prone simply by using bad default settings. Solaris uses a default prefix length ("netmask") setting of /128 for IPv6 addresses even though RFC 4291 and its predecessors explicitly state that the prefix length for all but a few special address ranges is always /64. Configurations which appear to be correct, but aren't, can easily confuse the enemy for some time.

The Solaris installer creates an `/etc/hosts` file which assigns the name `loghost` to the IPv4 loopback address. In an environment that uses a central log host and a name server, this leads to an inconsistent configuration until the enemy fixes either the resolver configuration or the `/etc/hosts` file. If it has also enabled IPv6 support on the system, then the same procedure repeats with `/etc/inet/ipnodes` for the IPv6 loopback address.

Another tactic is generously applied by most Unixen: The hostname is often spread all over `/etc`, making it a tedious job to just rename a single machine.

Configuring a Fedora Core 2 box as an IPv6 router shows how to use an inconsistent configuration syntax to confuse the enemy: In `/etc/sysconfig/network` the lines

```
NETWORKING_IPV6=yes
IPV6FORWARDING=yes
IPV6_ROUTER=yes
```

show how mixing underscore and no-underscore notation and prefix and postfix category naming can be easily applied to confuse the enemy admin. Of course, if the file was read by a proper parser, then it could flag parsing errors if the keywords were misspelled. But this file is a shell script, so if one of the variables is misspelled there won't be any visible problems—except that the router doesn't work as expected.

Finally, Solaris 10 introduced the "service management facility" (SMF) as a substitute for `init` and the SysV-style init scripts. It speeds up the boot process, deals with dependencies between services and is far nicer to handle than traditional init scripts—until the enemy needs to change the settings for a service: Then it will face a configuration "data base" that keeps binary representations of lengthy XML-based "service manifests". Editing an XML file with `vi` is great fun to watch the enemy do, especially if it is sitting at a VT100 terminal trying to solve a problem that just brought a system down.

The beauty of all these tactics is that they appear nothing more than a slight awkwardness. If the enemy loses time because of them, possibly at 50 000 € per minute, then that isn't our fault, really. And it *will* lose time because of them. Lots of time.

## Change handling and Upgrades

All tactics so far deal with a "static" software that doesn't change over time. In practice, virtually all software is continuously changed, updated and extended. We can use this to hit the enemy when it hurts a lot: during upgrades.

Again we can learn from various hardware vendors who sell different components under the same name. If you have ever seen the enemy trying to replace a broken network card with a new card of "the same model" you know the beauty of this plot:

Of course the new card doesn't work, so the enemy assumes that something else is the problem. After replacing virtually everything else in the machine it might finally "re-install the driver" from the CD-ROM shipped with the new card and surprise, afterwards the machine works again—until the next major upgrade is mass-deployed to all machines and this one with the "same model" network card loses its network connectivity in the process. The loss of service may sum up to several days in the long run.

The same can be done with software. The beauty of the BSD ports collections is that they usually obtain the source files from the original source of the software. If a source file is kater on updated there, then the ports collection makefiles will reject that source file as "broken" since it doesn't match the stored checksum. To rebuild a working system the enemy will first have to upgrade its ports tree and then recompile all and everything. If the timing is right, there will be new releases of KDE, GNOME, OpenOffice and a variety of other large packages, causing all sorts of minor problems that are tedious and therefore expensive to fix.

The BIND name server package did a very sneaky trick with the upgrade from 8.2.2 to 8.2.3: They called 8.2.3 a "maintenance release" that didn't change any functionality but just contained bugfixes. Which was perfectly true, except that they changed the way `dig` handled the `-x` option with IPv6 addresses: Instead of looking up the address in the (by then deprecated) bitlabel format, they changed it to use the (by then re-established) nibble format. The results to various shell scripts using `dig` to query the DNS could have been most satisfying; unfortunately it was way too early for pulling this trick, so only a single IPv6 advocate writing a book on the topic was seriously affected.

In a project that I was involved with some years ago, the closed-source software vendor forced the enemy to upgrade to a new release. The software was meant as a micro-billing system but the enemy used it as a means to gather statistical data only. The upgrade involved various reorganizations of the data bases which the vendor claimed not to require any additional disk space. The script they supplied to the enemy to do the reorganization was expected to run two weeks, during which the system couldn't gather any additional billing data. Most unfortunately, the enemy employed a single better-than-average application admin which managed to first split the script into about twenty separate steps. Taking data base dumps after each step it was able to revert to the results of the previous step after the occasional failure. In one case it temporarily doubled the disk space by acquiring a second external storage array before continuing. The entire upgrade took almost five weeks; afterwards said admin left the project. If the enemy had actually used this system for its billing, then this plot would have sent it straight into bankruptcy.

As these examples show, it is both feasible as well as worthwhile to drive the enemy into upgrade hell. Watching it squirm, trying to delay the upgrade while it fully realizes that it can't possibly escape it in the long run, is one of the most satisfying experiences in every software developers career.

## Security Aspects

Of course, excellent software is also secure. The Sony "no other notebook brands are affected" line concerning their exploding batteries is about as inapplicable as Microsoft's "our primary goal now is to improve security" while they still ship their Internet Explorer with ActiveX support and Office with VBA. So how can security aspects help to deter the enemy?

There is a technical tactic that is applicable: We can add user-configurable support for inherently insecure features, like support for SNMP "set" operations; this leads back to the configurability topic.

As a psychological trick we can simply deny all alleged security problems in a manner that shows the enemy that we feel personally offended by its allegations and no, there is no security issue at all. Then if there ever *is* an issue, then the enemy will deeply distrust the security of our software.

## Incompatibilities

Most of todays datacenter environments are highly heterogeneous. Incompatibilities, ranging from straight non-interoperability down to slight variations between different Unixen make life more difficult to the enemy while there is no single Unix to blame.

Several years ago I witnessed the enemy buying three PCI cards of different types to be put in a single machine. Any two cards worked together flawlessly, but all three together rendered all available test machines unbootable. All three card vendors blamed the others on the problem and refused the enemy a refund for their respective card.

Back in 1998 a Solaris NFS server and a Linux NFS client would manage to transmit approximately 100 kB/s between each other. Solaris-Solaris or Linux-Linux setups would easily do 1–3 MB/s. Unfortunately the enemy found ways to tune the Linux client in a way that raised the throughput quite significantly, but still the performance was less than satisfying.

Simply using different option letters for the same command makes scripting difficult to the enemy and can wreak the occasional havoc that the enemy admin fears: The `netstat` command with both Solaris and the BSDs uses an option `-f inet6`

to request IPv6-related information; with Linux it is `-A inet6`. Various Solaris commands like `df` use the `-h` option not to display a short command synopsis but to render their output in a "human readable" style.

So far these incompatibilities are mere nuisances to the enemy. But a Solaris box in a mostly BSD-derived environment will eventually embarrass the enemy admin with a gem that looks basically like

```
# /usr/sbin/shutdown -g 10 -i 5 -y \
    Shutting down in ten minutes
System going down in 10 seconds
```

The ultimate incompatibility example is the `killall` command. With Linux and the BSDs it kills all processes with a given program name but with Solaris and other true SVR4 Unixen it will do what its name suggests—it will kill *all* processes, shutting down the system the hard way. Linux has a command `killall5` which behaves like the SVR4 `killall` and Solaris has a command `pkill` which behaves similar to the Linux/BSD `killall` command. Eventually the enemy will learn to be very careful about the `killall` command; usually it will learn the hard way.

## Error Handling

When an error occurs within our software, be it an internal problem or bad input or a problem with the environment it runs in, like a full file system, then we can apply some more tactics.

If we didn't care about the excellence of our software, then we could simply send a cryptic error message of the "General Protection Fault" style to the enemy and then ask it to decide if it wants to "Abort, Retry or Continue?" Especially if we don't define a default decision for the enemy it will feel completely lost.

We can also write all logging entries into a single file, mixing debugging information, notices and more or less serious error conditions into a single file. If we don't include time stamps and use multi-line messages that aren't even separated by newlines, then the enemy will have great trouble to monitor this log file. If we use standard block-buffered I/O for this file "for performance reasons" and "forget" to open the file in append-only mode, then the enemy can't add periodic timestamps to the file, the most recent entries are either missing or incomplete due to buffering and if the writing process crashes, then the most vital last entry is effectively unavailable for cleaning up the mess. The billing software mentioned above has proved that this approach is both applicable and effective; unfortunately the effects on the excellence of our software usually forbid this tactic. Excellent software uses the `syslog(3)` API and even sets the priorities of each log message to a reasonable value.

Similarly, using the `assert()` macro or simply dumping a Java backtrace is generally considered bad style. The `rsync(1)` command tries to flood the enemy with information it can't use; a full file system at the destination first provides an "XXX write failed, filesystem is full" message and then follows up with seven additional lines that the average enemy can't understand, including the source file names and line numbers where the related secondary errors occurred.

My personal favourite in this category is Debian and how it deals with the IPv6 configuration in `/etc/network/interfaces`. As mentioned above the prefix length is effectively always 64 bits. So a configuration like

```
iface eth0 inet6 static
        address 2001:db8:fedc::1
```

provides all the information necessary to configure interface `eth0` for IPv6. Still, the `ifup` command will complain that

```
Don't seem to be have all the
variables for eth0/inet6.
Failed to bring up eth0.
```

Only after adding a line `netmask 64` to the configuration will the interface configure. The artful combination of a grammatical mistake in the error message, explicitly demanding a constant to be configured, using an IPv4 term "netmask" for an IPv6 prefix length and finally denying any hint at the problem in the error message will make it absolutely plain to the enemy that it isn't welcome to use this software.

## High Risk User Interfaces

Finally, the ultimate weapon against the most stubborn enemy is a high risk user interface. This is the software equivalent of a gun without a safety catch.

This tactic is commonly cloaked by asking the enemy for explicit confirmation for even the most simple operations: "Do you really move this file to trash?" asked for every single file out of five hundred will quickly teach the enemy to confirm whatever the system asks. Not only will this result in the same net effect as not asking for any confirmation at all, it will also annoy the enemy and, most importantly, it will make it confirm whatever really dangerous operation it accidentially invokes. Exposing the enemy to Microsoft Windows will quickly make it reach this "whatever it asks, just hit Return" mentality.

Solaris 10, 06/06, first made the excellent zetta file system (ZFS) available to the public. ZFS manages multiple file systems within a storage pool. To remove such a file system from a pool, the command

`# zfs destroy` ⟨file system⟩

will destroy a file system without further confirmation. But there's more to it: ZFS supports snapshots, which are named ⟨file system⟩`@`⟨snapshot⟩. To remove a snapshot, the command is

`# zfs destroy` ⟨file system⟩`@`⟨snapshot⟩

and again it doesn't ask for confirmation. It is only a matter of time until the enemy wants to release a snapshot but accidentially nukes an entire file system. This example more than compensates the lack of even the remotest hint of subtlety with the enormous degree of devastation it can cause.

The beauty of high risk user interfaces is obvious: Whatever happens, it is the enemies fault, not ours or that of our software.

## Summary

We have seen that there is a wide choice of low-effort, devastating-impact tactics to discourage even the most stubborn enemy from using our excellent software. Most of them can be made to appear "accidential" or "slightly awkward" rather than intentional and malicious.

With these weapons available and properly understood we can easily teach the enemy never to use our excellent software (twice).

## About the Author

Benedikt Stockebrand is a BSD-biased "generic Unixer" with a strong background in system administration and large-scale data center design and operation. He is working as a freelance trainer, author, IT journalist and consultant with a current focus on IPv6 operations.

He has been repeatedly charged with offensive sarcasm but so far escaped conviction.