Design of the Ocotillo PRNG

Eric Lee Green

September 15, 1999

Abstract

Most pseudo-random number generators have a predictable sequence and limited number of seed values, making them useless for cryptographic purposes. Described is a pseudo-random number generator which is purported to be cryptographically secure once seeded with appropriate sources of randomness. Standard cryptographic components (MD5, TwoFish) are used along with whatever sources of randomness are available in order to produce hopefully unpredictable results.

Contents

1 Legal Boilerplate

The MD5 message digest algorithm implementation used is copyrighted by RSA and was taken from an Internet RFC. The TwoFish implementation is copyrighted by Dr. Brian Gladman. Both are covered by a BSD-style license that allows free use as long as their source is properly attributed.

All other portions of Ocotillo are Copyright 1999 Enhanced Software Technologies Inc. (the makers of BRU, the #1 tape backup solution for Linux and Unix), and is also covered by a BSD-style Open Source license. My thanks to EST (http://www.estinc.com) for allowing me to release this work to the general public.

This paper is Copyright 1999 Enhanced Software Technologies Inc. Again, my thanks to EST for allowing me to release this work to the general public.

2 Rationale

Strong network authentication and encryption requires a strong source of key material. If keys are predictable, then a potential attacker does not need to try all possible keys in order to break into your system or intercept sensitive materials.

The original motivation for Ocotillo arose due to two factors:

1. The lack of a good PRNG as a standard component on the majority of Unix platforms that my employer supports, and

2. the poor cryptographic quality of the 'whrandom.py' module that is the "best" PRNG that comes with the Python programming language.

FreeBSD and Linux both have a /dev/random which produces hopefully "truly" random numbers. The majority of Unix platforms that EST supports do not contain such a source of true random numbers. The only standard there is the 'rand' PRNG.

Examination of the 'rand' source code under FreeBSD, which presumably is closely related to its implementation on other Unix operating systems that share the same heritage, shows that this PRNG is utterly predictable. If you know one output, you can guess all future and previous outputs. Thus while it produces a statistically random distribution, as a cryptographic component it is an utter failure. In addition, with only 32 bits of initial randomness, using it to produce even 56-bit session keys for export-strength DES is rather futile.

The whrandom.py PRNG under Python produced a statistically random distribution, but similarly failed the predictability test. There are only 24 bits of initial randomness. Assuming that keys are generated shortly after the PRNG is initialized, keys can thus be brute-forced in much less time than would be theoretically possible for a key of a given length larger than 24 bits.

What this examination showed was that statistical randomness is not enough for a cryptographic pseudo-random number generator. Rather, the most needed property is **unpredictability**. A random distribution of outputs is useless if the outputs are predictable and can thus be used to gain important information about keys generated with the PRNG.

3 Design criteria

The design was primarily influenced by *Yarrow* (http://www.counterpane.com), and its name (Ocotillo) is in homage to that influence (ocotillo is a plant of the Arizona desert where EST is located). Design criteria included:

1. *Use of standard cryptographic components.* I do not have any delusions about my abilities as a cryptographer. Relying upon tested components that have undergone extensive crytoanalysis by a variety of cryptographers should at least insure that any flaws in my implementation are not flaws in the underlying cryptographic components.

2. *Address the attacks mentioned in the Yarrow paper.* The Yarrow paper mentions many attacks against cryptographic PRNG's. My hope is to address most of those attacks.

3. *Run in user space with no kernel modifications needed.* The software using this PRNG will hopefully be mass-deployed within Fortune 500 companies via automated software distribution tools. These tools cannot and will not install Unix kernel modifications. In practice, this design criteria is what produced the biggest problems, and which also

results in Ocotillo's biggest failings in contrast to Yarrow (which burrows deep into the Win32 kernel for its sources of entropy).

4. *Run in unattended mode.* It must be capable of operating even if the machine has no keyboard and no way of interacting with a user other than via the network (and note that network events can be monitored and thus are rather shaky as a source of pure entropy). This is by far the hardest part of the design, and the reason for most of the possible attacks upon the design. If non-public interaction is possible, additional entropy can be fed into the PRNG in order to increase its unpredictability.

5. *Be capable of producing 128-bit session keys.* Analysis of our application showed that 128-bit encryption would be adequate for the near future, unless some unforseen attack against the winner of the AES competition arose. If you have sufficient sources of randomness such that you can regularly re-key Ocotillo you may be able to safely generate 256-bit session keys, but I would conservatively characterize it as having only 129 bits of security.

6. *Be able to use any of the AES contenders as its core block cipher component.* We intend to support whatever algorithm wins the AES contest as our standard method of encryption. If a block cipher is at the core of the PRNG, it makes sense from a code re-use perspective to use the same block cipher as is used to encrypt the actual data.

Note that speed was not a criteria. The needs for key material in this application are modest, and it does not matter that it may take several seconds to initialize the PRNG when the system initially boots.

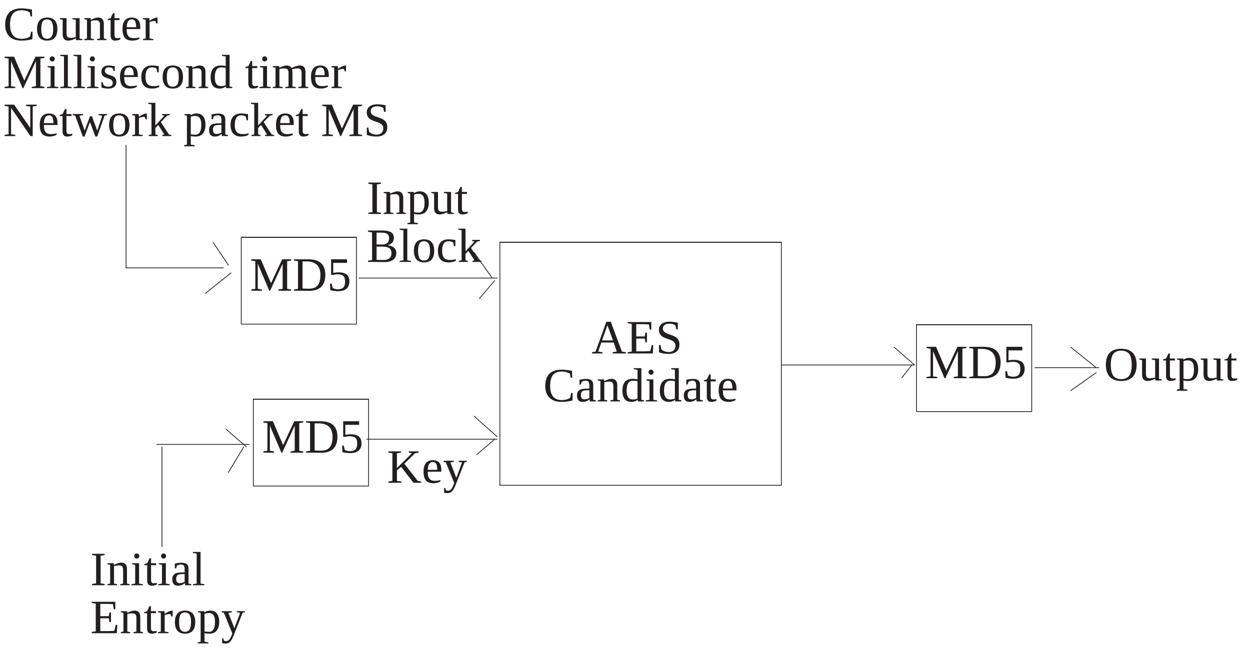
4 Design

4.1 Overall design

MD5 was chosen as the entropy stirring algorithm due to the fact that its 128-bit output matches the 128-bit block size and encryption key size of the AES candidate chosen.

Any AES candidate can be used for the block cipher. RC6 was used for the original prototype, but due to patent considerations, Twofish was substituted for the initial release. The change took approximately fifteen minutes.

These components are tied together as follows:



All blocks marked MD5 are MD5 "pools". That is, when the current value is desired, they are copied to a new MD5 context, the current value extracted from that context, while the original context is retained to put future values into.

The AES candidate is operating in block encryption mode. That is, it uses the key and input block and produces an encrypted output.

4.2 Initial entropy

Initial entropy is fed into the MD5 key pool until it is judged that sufficient entropy has been accumulated. The mechanism is to feed as much stuff as possible into the MD5 and allow it to "distill" any randomness out of the input. This will produce some attacks, mentioned later in the "Attacks" section.

Some sources of initial entropy:

1. The current millisecond timer.
2. The current contents of the system log files, files which change over time and are presumably unpredictable by an outside attacker. Note that this is one way to attack the algorithm, though in our case it doesn't matter (if you have 'root' access, the system is already compromised).
3. The output of "netstat -i". The number of packets that have passed through the network interfaces since the machine was booted are of particular interest. Rather than attempt to extract this information (netstat's format changes wildly from platform to platform), I feed the whole thing into the MD5 hash in hopes that the MD5 will "distill" the randomness out of the output.
4. the output of the 'ps' command. This is not very random, since the list of commands should be the same every time we boot, but the low-order milliseconds of some of the times should be randomly different by one bit every time it runs (detirminism is nice, but the system state cannot be 100% detirmined without exact knowledge of exact state of every portion of the system).
5. the output of 'uptime'. This may be predicted if the attacker has access already to our system, but if our attacker is outside, this adds a few bits of randomness.
6. The clock time of my process (this may be as little as one bit of randomness, depending upon when the PRNG is initialized).
7. The millisecond timer after all of the above has taken place.

Occasionally, when sufficient entropy has become available, it will be hashed into the MD5 pool for the key, and the AES block cipher will be re-keyed. Calculating when it is advantageous to do this is a Hard Problem. Ideally it should only be done when 128 bits of true randomness have been accumulated. Overestimating the amount of true randomness is easy to do, especially in the face of attacks. At the moment I have not implemented this part of the algorithm.

4.3 Input entropy

As entropic events become available, they are hashed into the MD5 pool on the input block. These events may be things like:

1. The lower-order bits of the millisecond timer at the time that a command comes in to the program,
2. The lower-order bits of the millisecond timer at the time that a network packet comes in (note: a flood attack may compromise this),
3. The lower-order bits of the millisecond timer at the time that a key is pressed (I don't have a keyboard in my application, but others may).

4.4 Normal Operation

When 128 bits of output are requested, the following takes place:

1. A counter is incremented and fed into the MD5 pool on the input block. This keeps the input block incrementing even if no entropic events have taken place.
2. The current millisecond timer is fed into the MD5 pool on the input block. This will add any randomness to that pool.
3. The current value of the input MD5 pool is obtained.
4. That value is fed to the encrypt() function of the block cipher.
5. The output of the block cipher is fed into the MD5 pool on the output.
6. The current value of the output MD5 pool is obtained.
7. That value is returned as the value of the PRNG.

5 Attacks

5.1 Attacks upon the block cipher

Some block ciphers may be succeptible to differential analysis or similar attacks. This is especially true if attacks upon the inputs result in a known plaintext on the input block for the block cipher.

The MD5 hash on the output is a recent addition to Ocotillo. One of its purposes (but not the only one) is to prevent direct attacks upon the block cipher. Since the ciphertext is never directly seen, attacks which rely upon direct access to the ciphertext should fail.

5.2 Entropy attacks

These attacks are upon the sources of entropy for the block cipher. Some attacks could be:

1. Dictating or duplicating the initial seed. If one can control exactly what the outputs of 'ps', 'netstat -i', etc. are, as well as stopping the timer, then one can predict the initial key value and thus presumably with other attacks be able to predict the outputs of the PRNG. Note: On Unix, this requires root access. Game Over.

2. Insufficient entropy for the initial seed. If insufficient entropy has been gathered, the attacker may be able to guess possible values for the initial seed and break the PRNG in less time than would be theoretically possible. This is the most worrisome attack.

3. Stopping the timer. If done during normal operations, the block cipher essentially operates in block counter increment mode. If done as part of an attack that relies upon dictating the initial seed, it could be disasterous. But note: On Unix, this requires root access. If one has root access, then the game is over anyhow (the whole point of the crypto component of this application is to prevent people from obtaining root access).

4. Flooding the PRNG. Presumably if one floods the PRNG with requests, the block cipher is essentially operating in block counter increment mode. The cipher should itself be fairly resistent to crack even under those circumstances, but prevention of flood has to be handled at the applications level. For this particular application, the PRNG API is not exposed to outsiders, and the protocol is resistant to flood attacks (such as, e.g., floods of session re-key requests). For other applications, flood attacks may be a more critical issue.

5.3 State attacks

These are related to entropy attacks, but presume that you have obtained the key and/or input value of the cipher at some point in time. You will then use this to predict future or past outputs of the cipher based upon the fact that the input value of the cipher has limited entropy (i.e., is basically operating in counter mode, especially if combined with an entropy attack).

One purpose of the MD5 on the output is to make the algorithm resistant to state attacks. Even if you know what the current input values are, you cannot necessarily detirmine what the output value will be, due to the fact that the MD5 on the output has essentially distilled all of the prior entropy fed into it. The output MD5 state is never saved to disk other than possibly to swap. Under Unix, access to swap is similar in difficulty to access to internal state, i.e., one must have root access (Game Over for our implementation).

Some state attacks, and how Ocotillo may (or may not) be succeptible:

1. Backtracking attacks: These use knowledge of the current state in order to find or guess prior values. If combined with an entropy and/or flood attack, so that the values on the input counter are easily guessable, the state going into the output MD5 can be predicted with varying degrees of success for near-past, worse degrees of success for far-past. Assuming that a few bits of entropy are continually being added to the input and thus filtering into the output, the output MD5 hash value will not be easily predictable even here, due to the fact that it has accumulated all of that past randomness and incorporated it into its output. The MD5 algorithm's output state is presumably influenced by all past randomness, not just the randomness of the last few samples.

2. Predictive attacks: If the state has been compromised at some point in time, predictive attacks that predict future output of the PRNG are more worrisome than backtracking attacks for the Ocotillo design. The output of an MD5 hash algorithm can be assumed to be directly related to its internal state. The past history needed to arrive at that internal state is not known, but if it is fed another value, presumably its output when fed that value can be predicted by setting up an identical MD5 routine with identical state and similarly feeding it the predicted value. The MD5 thus does not add any strength in this instance. The ability to predict output depends entirely upon the amount of entropy being sent into the cipher. Our sources of entropy are very limited,

and re-keying can be considered to never happen in normal use (which requires results numbering in the dozens, i.e. not enough to gather enough entropy to re-key the cipher and thus re-set to an unknown state). Thus this can be considered a weakness of Ocotillo.

3. Meet-in-the-middle predictive attacks: If the state has been compromised TWICE at some point in time, this attack attempts to find values that occurred between the two compromises. Ocotillo's weakness when faced with predictive attacks will be the biggest problem here. The second compromise will mostly just allow the attacker to know when he has "guessed right" about the sequence that he predicted in the predictive attack. The only solution here would be more entropy, which is not available in this particular deployment environment.

6 Conclusions

It is apparent, from the research that went on in writing Ocotillo, that a PRNG that runs entirely in user mode will never be entirely secure. The basic problem is entropy, or lack thereof. This is the one area where Ocotillo most needs help, and is the one thing that would most compromise Ocotillo in actual deployment. An attacker who manages to obtain the state of the various system parameters used to initialize Ocotillo at boot time, or manages to simulate that by running a parallel system of identical configuration, may be able to compromise the initial seed and thus greatly impair the strength of the PRNG.

It appears that Ocotillo will satisfy our particular criteria. The only known full compromises require root access on the machine that Ocotillo is running upon. Since the purpose of the cryptographic component of which Ocotillo is a part is to prevent root access on the machine that Ocotillo is running upon, in that regard Ocotillo can be viewed as a success. As a general-purpose PRNG, on the other hand, Ocotillo could be considered a success only if augmented with additional sources of entropy. It is not recommended, for example, that unmodified Ocotillo be used as the key generator for a personal encryption product. Augmenting its inputs with a source of true entropy would be the only appropriate action in that case.

The best recommendation is that if your operating system provides a source of true randomness such as /dev/random on Linux or FreeBSD, you should either use that directly or use that to seed and occasionally re-seed Ocotillo. The second-best recommendation is that if your operating system vendor does not have a true random number generator such as /dev/random, you should petition them for one – a generator which burrows deep into the internals of the OS can take advantage of far more timing events to generate random bits. If all else fails, Ocotillo may fill your needs – but you should carefully evaluate the predictability of its sources of randomness and, whenever possible, feed your own randomness into it in order to offset the impact of predictive attacks upon the algorithm.