When Malware Authors Study Algebra: The Group Theory Inside Bedep's DGA

By Threat Research TeamThreat Research Team

Archived: 2026-05-06 02:00:47 UTC

TLDR: Bedep was a malware family active mainly in 2014 and 2015 that used an unusually sophisticated domain generation algorithm, or DGA, to hide its command-and-control infrastructure. Instead of relying only on the date, it used real foreign exchange rates published by the *European Central Bank*, making its future domains much harder to predict in advance. Under the hood, the malware used ideas from group theory to generate a fixed set of unique domains without collisions, a rare example of advanced mathematics being applied directly in real-world malware.

Most domain generation algorithms are boring. A linear congruential generator, a date seed, some modular arithmetic, a character table -- `seed = (seed * 0x41C64E6D + 0x3039) % 2^31`, rinse, repeat. You reverse them in an afternoon, pre-compute next month's domains over lunch, and move on.

Bedep author implemented a number theory textbook inside a 32-bit DLL, and the result is one of the most mathematically elegant pieces of malware ever shipped.

This post is a deep dive into this unique engineering.

What is Bedep?

Bedep is an ad-fraud botnet that was active from late 2014 through 2015, delivered exclusively through the Angler exploit kit. When a Flash zero-day (CVE-2015-0311) was burning, Angler was dropping Bedep. The malware itself was a modular downloader focused on click fraud, but its infrastructure is what made it interesting. In early 2015, new variants appeared with a DGA for C2 resolution. A three-day sinkhole by Arbor Networks (ASERT) caught phone-homes from roughly 82,000 unique IPs, spread across every continent except -- notably -- Russia.

Dennis Schwarz originally reversed the DGA at ASERT, who published a proof-of-concept Python reimplementation and a detailed write-up titled *"Bedep's DGA: Trading Foreign Exchange for Malware Domains."* The original blog post has since been taken down from NETSCOUT's site, but survives on the Wayback Machine. The PoC lived at `github.com/arbor/bedep_dga`. Dennis got the algorithm working but noted that parts of it were opaque -- he called the core transform a "blackbox" and flagged the embedded table as "likely something Fermat number related." He was right. Let's open that box.

The ECB trick: Seeding a DGA from the financial markets

Before the math starts, Bedep needs a seed that both the bot and the botmaster can independently derive without communicating. Most DGAs use the date. Bedep uses the date *and* the global foreign exchange market.

The algorithm fetches two XML files from legitimate public services:

1. **UTC time** from `earthtools.org/timezone/0/0` -- the current timestamp, converted to "days since year zero."
2. **Euro foreign exchange reference rates** from `www.ecb.europa.eu/stats/eurofxref/eurofxref-hist-90d.xml` -- the European Central Bank's daily publication of EUR conversion rates against 31+ world currencies, updated every business day.

The time is used to select which day's rates to use. Specifically, Bedep picks "last Tuesday's" exchange rates -- meaning the preceding week's Tuesday until Thursday, after which it rolls forward to the current week's Tuesday. Only rates falling on a Monday boundary (in the algorithm's day-counting scheme) are considered. This gives a roughly weekly rotation with a deterministic selection rule.

From the chosen date, Bedep extracts up to 48 currency rates -- USD, JPY, GBP, CZK, BGN, HUF, and so on. Each rate is parsed through a slightly broken custom `atof()` implementation (off by a bit in the least significant digits -- a quirk, not a feature), packed into a 64-bit IEEE double, and its low 32-bit dword is extracted for use in the algorithm.

Here is what a single run looks like, from the PoC output for April 7, 2015:

```
parsed 31 currencies from 2015-04-07 (currency date):    USD: 1.0847      JPY: 130.33     BGN: 1.9558
   CZK: 27.455      DKK: 7.4714    GBP: 0.7286    HUF: 299.08      PLN: 4.0578    RON: 4.4165    SEK:
9.374       CHF: 1.0438    NOK: 8.73    HRK: 7.619       RUB: 59.8265   TRY: 2.8079    AUD: 1.4192
 BRL: 3.3979    CAD: 1.3563    CNY: 6.7241      HKD: 8.4086    IDR: 14091.77    ILS: 4.267       INR:
67.598    KRW: 1183.28    MXN: 16.1919     MYR: 3.9527    NZD: 1.4423    PHP: 48.3        SGD: 1.4724
   THB: 35.335    ZAR: 12.8345
```

Why is this clever? Because the ECB rates are:

- **Publicly available** -- anyone can fetch the same XML
- **Globally consistent** -- every bot on every continent gets the same data
- **Unpredictable in advance** -- nobody can predict next Tuesday's EUR/USD rate
- **Historically verifiable** -- the ECB publishes archives going back years
- **Impossible to tamper with** -- the ECB is not going to modify their reference rates to help you sinkhole a botnet

And because the malware fetches this from `ecb.europa.eu`, the request looks like legitimate traffic. A Snort signature (SID 33188) was written to detect it, but it fired constantly on real users checking exchange rates.

The mathematical core

This is where Bedep stops being a clever botnet and starts being a discrete mathematics exercise.

Precomputed subgroup parameters

The DLL embeds a lookup table -- 102 entries of 8 dwords each, stored as JSON in the PoC (`transform2_table_varN.json`, one per config variant). At first glance it looks like random constants. It is not.

The `transform2` function takes a hash derived from the exchange rates and days-since value, then uses it as an index into this table. Each table entry, after being decoded through config-specific XOR and multiply constants (`0x663d81 * entry[0] ^ value3` and `0x281 * entry[1] ^ value2`), yields two critical values:

- **p** -- a prime number
- **q** -- an integer such that q divides p - 1

This is the subgroup structure. The table stores primes whose group orders (p - 1) have known factorizations with small prime factors. The malware doesn't need to factor p - 1 at runtime -- that hard work was done offline by the author and baked into the DLL. Dennis Schwarz noted these were "likely something Fermat number related," and he was onto something: the primes are chosen so that their totient has a smooth factorization, making generator-finding tractable.

The number of domains to generate comes directly from q - 1. For the first config, this works out to 22 domains; for the second, 28 -- totaling 50 domains per weekly rotation.

Finding generators: A textbook primitive root search

With p and the prime factorization of p - 1 in hand, Bedep needs a generator of the multiplicative group ((\mathbb{Z}/p\mathbb{Z})^*) -- or more precisely, of a specific subgroup of order q.

The `transform8` function does this in two stages. First, it runs a **trial-division prime sieve**: starting from 3, stepping by 2 (skipping evens), it tests each candidate against all previously found primes. It collects up to 37 small primes, then filters for those that actually divide q (the subgroup order). This gives the prime factorization of q.

Then `transform3` finds the **primitive root** itself. This is the standard algorithm straight from a number theory textbook:

```
 def find_primitive_root(p, prime_factors_of_order):    for g in range(2, p // 2):        is_generator
= True        for q_i in prime_factors_of_order:            # If g^((p-1)/q_i) ≡ 1 (mod p), then g
generates            # a proper subgroup, not the full group. Reject it.            if pow(g, (p - 1)
// q_i, p) == 1:                is_generator = False                break        if is_generator:
       return g    return None
```

The actual code in `transform3` is this logic rendered in C-style loop structure with manual index tracking, but the mathematics is identical. For each candidate g starting from 2, it checks:

[g^{(p-1)/q_i} \not\equiv 1 \pmod{p} \quad \text{for all prime factors } q_i \text{ of } p-1]

If g passes all checks, it generates the full group. This is literally the algorithm from Section 11.1 of Shoup's *A Computational Introduction to Number Theory and Algebra*, implemented in a malware DLL.

Walking the cyclic group

Now comes the payoff. With a prime p, a generator g, and a subgroup order q, the `transform7` function sets up the DGA's iteration state:

```
modulus    = p                          # field_c seed       = pow(g, random_e, p)        # field_4 -
- starting element step       = pow(g2, random_f, q)       # field_8 -- step exponent
```

The starting position `seed` is g raised to a random exponent (derived from `rdtsc`, the CPU timestamp counter). The step is similarly randomized from a second generator. Both are computed via modular exponentiation.

Then, for each domain, the iteration is simply:

```
seed = pow(seed, step, modulus)     # one step through the cyclic group domain = seed_to_domain(seed,
currencies)
```

That single line -- `pow(seed, step, modulus)` -- is the entire DGA engine. Each call advances the state to the next element of the cyclic subgroup.

Because the subgroup has order q, and step is coprime to q (guaranteed by the generator construction), this walk visits every non-identity element of the subgroup exactly once before cycling. The number of domains generated equals q - 1, which is precisely the number of non-identity elements. No collisions. No wasted iterations. No off-by-one hoping you hit enough domains.

The key insight: the set of group elements visited is deterministic (fixed by p, g, and the exchange rates), but the *order* of visitation depends on the random exponents from `rdtsc`. Every infected machine walks through the same set of q - 1 elements, generating the same q - 1 domain names, but potentially in a different order. Same destinations, different paths.

```
flowchart TB    subgraph cyclic_group ["Cyclic Subgroup of (Z/pZ)*"]        E1["g^1 mod p"] -->
E2["g^2 mod p"]        E2 --> E3["g^3 mod p"]        E3 --> E4["..."]        E4 --> Eq["g^(q-1) mod
p"]        Eq --> E1    end    subgraph walk ["DGA Walk (order depends on rdtsc)"]        S1["seed_0"]
-->|"pow(s, step, p)"| S2["seed_1"]        S2 -->|"pow(s, step, p)"| S3["seed_2"]        S3 --
>|"pow(s, step, p)"| S4["..."]        S4 -->|"pow(s, step, p)"| SN["seed_(q-2)"]    end    S1 -.->|"=
g^e mod p\n(random start)"| cyclic_group    S2 -.->|maps to| E3    S3 -.->|maps to| Eq
```

From Group Elements to Domain Names

Each group element is a 32-bit integer. Turning it into a domain name is the final step, handled by `transform11`. This is less mathematically elegant and more of a traditional mixing function, but it has a nice property: every character position in the domain uses a *different* currency rate.

The algorithm cycles through the full list of parsed currencies (31 in the example above). For a domain of length 18, that means character 0 is influenced by USD, character 1 by HRK, character 2 by MXN, character 3 by GBP, and so on. Each character is computed by:

1. Multiplying the currency's 3-letter name (packed as a 32-bit int) and its 64-bit floating-point rate by large constants

2. XORing and shifting the result with the current seed and the days-since value
3. Taking the result modulo 26 (for positions 2+) or modulo 36 (for the last two positions, allowing digits)
4. Adding the ASCII offset for 'a'

The domain length itself is derived from the seed XORed with a rate-dependent value, bounded between 12 and 18 characters. All domains use the `.com` TLD.

A trace from the PoC output shows the mixing in action:

```
seed: 0xa13c9652    mixing in USD's rate    mixing in HRK's rate    mixing in MXN's rate    mixing in GBP's rate    ...    mixing in ZAR's rate    domain: rrpohktjlscncqxvt3.com seed: 0x558af439    mixing in USD's rate    ...    domain: wjavcjhazzxyxotkbi.com
```

Each domain is a fingerprint of the entire exchange rate vector, filtered through one specific element of the cyclic group.

The Full Pipeline

Putting it all together:

```
flowchart LR    ECB["ECB XML\n31+ currency rates"] --> DateSelect["Select last\nTuesday's rates"]    Earth["earthtools.org\nUTC timestamp"] --> DateSelect    DateSelect --> Mix["XOR/multiply\nrates + days_since"]    Mix --> TableLookup["Lookup table\n→ prime p, order q"]    TableLookup --> GenFind["Sieve primes,\nfind generator g"]    GenFind --> GroupSetup["seed = g^rand mod p\nstep = g2^rand mod q"]    GroupSetup --> Walk["Iterate:\nseed = seed^step mod p"]    Walk --> CharMix["Mix seed with\ncurrency rates"]    CharMix --> Domains["22-28 domains\nper config"]
```

Seven configs were observed in the wild, each with its own embedded table, XOR constants, and currency count (36 or 48). Each malware variant embedded two configs that ran per rotation -- the first generating 22 domains, the second 28, yielding 50 domains total. Across the observed campaign, domains were registered 2-5 days ahead of use through "Domain Context" registrar with Regway nameservers, all resolving to infrastructure at 5.196.181.244 and 46.105.251.1.

Why this matters

Compare Bedep's approach with its contemporaries:

Family	DGA Technique	Seed
Conficker.C	Date + simple arithmetic	Current date
Gamaredon	Nested character-range loops	Hardcoded ranges
Necurs	CRC32 hash of date components	Date + constant
Matsnu	Dictionary word concatenation	Days since epoch

Family	DGA Technique	Seed
Bedep	**Cyclic group walk mod p**	**ECB exchange rates**

Every other DGA on this list can be pre-computed arbitrarily far into the future. Bedep cannot -- because Tuesday's exchange rates don't exist until Tuesday. You can sinkhole Conficker's domains for next year during your coffee break. For Bedep, you have to wait for the ECB to publish, then race the botmaster to register the domains.

The mathematical guarantee is real: the cyclic group structure ensures exactly q - 1 distinct domains per rotation with no collisions and no wasted iterations. The primitive root construction ensures the generator produces every element. The precomputed tables avoid expensive factorization at runtime. The exchange rate seed provides unpredictable entropy from a trusted, globally-consistent, publicly-verifiable source.

This is not someone who copy-pasted a DGA from a forum. The author understood multiplicative groups modulo primes, knew how to find primitive roots, precomputed smooth-order primes offline, and wired the whole thing to the European Central Bank's daily publications. Whether they learned this from a university course, a textbook, or a cryptography library's source code, the result is unmistakable: this is applied algebra, deployed in production, at scale.

The original ASERT analysis is preserved on the Wayback Machine. The proof-of-concept implementation was published on ASERT's GitHub. The sample analyzed here is `e5e72baff4fab6ea6a1fcac467dc4351` (MD5) / `d0fb1b66b6e4da395892327be9f39adb4533e7759ace39f67bdde0bb1cdaef35` (SHA256).

Credits: The DGA was originally reversed by Dennis Schwarz at Arbor Networks / ASERT. This post builds on his work, focusing on the mathematical structure he identified but didn't fully unpack.

Threat Research Team

Threat Research Team

A group of elite researchers who like to stay under the radar.
